Pro Android with Kotlin

Developing Modern Mobile Apps

Peter Späth

Apress®

Pro Android with Kotlin

Developing Modern Mobile Apps

Peter Späth

Apress®

Pro Android with Kotlin: Developing Modern Mobile Apps

Peter Späth
Leipzig, Germany

ISBN-13 (pbk): 978-1-4842-3819-6 ISBN-13 (electronic): 978-1-4842-3820-2
https://doi.org/10.1007/978-1-4842-3820-2

Library of Congress Control Number: 2018955831

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484238196. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

To Margret.

Table of Contents

About the Author

Peter Späth, Ph. D. graduated in 2002 as a physicist and soon afterward became an IT consultant, mainly for Java-related projects. In 2016 he decided to concentrate on writing books on various subjects, with a primary focus on software development. With a wealth of experience in Java-related languages, the release of Kotlin for building Android apps made him enthusiastic about writing books for Kotlin development in the Android environment.

About the Technical Reviewers

Marcos Placona is a developer evangelist at Twilio and a GDE. He serves communities in London and all over Europe. He is passionate about technology and security and spends a great deal of his time building mobile and web apps and occasionally connecting them to physical devices.

Marcos is a great believer in open source projects. When he's not writing open source code, he's probably blogging about code on https://androidsecurity.info, https://androidthings.rocks, or https://realkotlin.com.

He's also a great API enthusiast and believes they bring peace to the software engineering world.

Massimo Nardone has a master of science degree in computing science from the University of Salerno, Italy, and has more than 24 years of experience in the areas of security, web/mobile development, cloud, and IT architecture. His IT passions are security and Android.

Specifically, he has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect.

He has also worked as a visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University), and he holds four international patents (in the PKI, SIP, SAML, and proxy areas).

He currently works as the chief information security officer (CISO) for Cargotec Oyj and is a member of the ISACA Finland Chapter board.

Massimo has reviewed more than 45 IT books for different publishing companies and is the coauthor of *Pro JPA 2 in Java EE 8* (Apress, 2018), *Beginning EJB in Java EE 8* (Apress, 2018), and *Pro Android Games* (Apress, 2015).

Introduction

The programs explained in this book, despite their strong affinity to the Kotlin way of thinking, will not be totally mysterious to Java developers or developers of other modern computer languages. One of the design goals of Kotlin is expressiveness, so understanding Kotlin programs requires little effort, even when the programs get shorter. But at some point, you have to pay for maximum brevity with a loss of expressiveness and a loss of readability.

When it comes to deciding what is better, I favor expressiveness over brevity, but be assured that a loquacious programming style is a no-go. In the end, professional developers want to write concise apps because less code means lower costs when it comes to maintenance.

The Transition from Java to Kotlin

Just to whet your appetite, you will now take a look at a really simple app—one that lacks a lot of features you would want to see in a more complex and professional app—and then rewrite it from Java to Kotlin. The app consists of a single activity and presents an edit field, a button, and a text field that reacts to button presses.

If you want to create it using Android Studio, you initiate a new project, disable Kotlin support, and edit the layout to contain a `TextView` widget, a `Button` widget, and an `EditText` widget. Then assign the IDs `edit`, `btn`, and `text` to them, respectively. The Java code is as follows:

```
package de.pspaeth.simplejava;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.*;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
```

```
        final EditText et = findViewById(R.id.edit);
        final Button btn = findViewById(R.id.btn);
        final TextView text = findViewById(R.id.text);

        btn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
               String entered = et.getText().toString();
               text.setText("You entered '" + entered +
                       "' and pressed 'Go'");
            }
        });
    }
}
```

Here are a few notes about the previous Java code:

- The `public` in front of the class says that it is visible from everywhere. It cannot be omitted here since otherwise the framework could not use the class.
- The `setContentView()` changes something by virtue of the "set," which is such a common construct that you might want to write it more concisely as `contentView = s.th.`instead, even with a variable of name "contentView" not actually existing or being private. A couple of competitor languages allow for this type of syntax. In Groovy for example, you can write contentView = s.th.and the language will internally translate it to `setContentView()`.
- The `final` in front of the three declarations is necessary in Java up to version 7 because the variables are going to be used in the anonymous inner class that comes a little later.
- Also, for the `setOnClickListener()` method, you might want to use `.onClickListener = s.th.` instead. It's the same for the `.setText()` a little later.
- The argument to `setOnClick-Listener()` is an object of an anonymous inner class; it is already an abbreviation of first declaring and then instantiating and using it. But you could be even more expressive with syntax like `btn -> do s.th.` or similar, just not in the Java language (well, at least not before Java 8).
- For the `et.getText()` method, you could just as well write something like `et.text`, which would express the same thing but is shorter.

A sister project that does the same thing but with Kotlin support is written in the Kotlin language as follows:

```
package de.pspaeth.simplekotlin

import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import kotlinx.android.synthetic.main.activity_main.*
```

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        btn.setOnClickListener { view ->
            val entered = edit.text.toString()
            text.text = "You entered '" + entered +
                "' and pressed 'Go'"
        }
    }
}
```

Looking at the Kotlin code more thoroughly, a couple of observations emerge:

- You don't need the semicolon delimiters. Kotlin checks at line breaks whether the statement is finished or whether the following line needs to be included.
- You don't need `public` in front of the class; `public` is standard in Kotlin.
- Instead of `extends`, you just write `:`, improving the readability a little bit.
- You don't need to specify `void` as a return type if a function doesn't return anything. Kotlin can infer that.
- Unfortunately, you cannot write `contentView = s.th.` as suggested earlier. The Groovy language, for example, allows for that. The reason why this can't be done in Kotlin is that the construct `contentView = s.th.` implies that there must be a class field named `contentView`, which is not the case. The compiler could check for appropriately named methods and then allow for that syntax, but the Kotlin developers decided to impose this restriction and to prohibit the construct if the field doesn't exist. The same is true for `setOnClickListener` because a field called `onClickListener` doesn't exist either.
- Instead of an anonymous inner class, you can use the functional construct `view -> ….` This is always possible if the addressed class, the listener in this case, just contains a single method, like `void onClick( View v )` in the base interface used here. The Kotlin compiler knows that it must use that particular single method of the listener class.
- The `EditText`, `Button`, and `TextView` variables no longer need to be declared. This is, however, not related to Kotlin but a mechanism provided by Android Studio. The import `kotlinx.android.synthetic.main.activity_main.*` brings you those fields automatically, derived from the resources.

To review, the Kotlin code with 559 characters does the same as the Java code with 861 characters. This is a savings of 35 percent, a percentage you can expect for more complex classes as well.

Despite the syntax being different from Java, the Kotlin compiler translates its source code to the same virtual machine bytecode as Java, so Kotlin can use the plethora of Java libraries that are out there in the wild, and Java developers switching to or also using Kotlin won't miss them.

This Book's Audience

This book is for intermediate to experienced Android developers wanting to use the new Kotlin features to address current Android versions and devices.

After reading this book, you will be able to use Android Studio and Kotlin to build advanced apps targeting the Android platform.

Being a Kotlin expert is not absolutely necessary for using this book, but having read introductory-level Kotlin books or studied online resources is surely helpful. The online documentation of Kotlin provides valuable resources you can use as references while reading this book.

Source

You can find all the code source shown or referred to in this book at https://github.com/Apress/pro-android-with-kotlin.

Online Text Companion

Some lists and tables, as well as some class and interface details, are available as part of the free source code download at https://github.com/Apress/pro-android-with-kotlin. References to such online resources are marked appropriately.

How to Read This Book

This book can be read sequentially if you want to learn what can be done on the Android platform, or you can read the chapters independently when the need arises while working on your Android projects. In addition, you can use parts of the book as a reference for both finding solutions to particular problems and determining how things can be done using Kotlin instead of Java. This book includes a description of special Kotlin language constructs that will help you make your code concise and reliable.

Specifically, Chapter 1 gives a short, bird's-eye view of the Android system. If you already have some experience with Android, you can skip it or just skim it.

Chapters 2 to 6 talk about the Android architecture's corner blocks: an application as a whole, activities, services, broadcasts, and content providers. If you are a pro-level developer, some of the information provided in these chapters might seem a bit basic and easy to find in the official Android developer documentation or elsewhere on the Web. The reason why I have included these topics is that the information in other sources is of varying quality—sometimes because of historical reasons, sometimes just because it is outdated. So, I tried to rectify some of these peculiarities and also provide you with a consolidated, fresh view on things. I hope I can save you some time when you get into the deeper-level nuts and bolts of Android work. You can also use these chapters as a reference in case you are in doubt about certain development issues while your Android project advances.

Chapter 7 briefly talks about the permission system. This is something you must of course be acquainted with if you develop pro-level Android apps.

Chapters 8 and 9 deal with APIs you can use in your app and user interface issues. Because both of these are big issues, it is not possible to mention everything that refers to these topics. I, however, will give you a selection of useful and interesting solutions for various tasks in these areas.

Chapters 10 and 11 take a deeper look at development and building strategies and describe how things can best be done inside Kotlin. While in the previous chapters the Kotlin code is presented in a more empirical way, in Chapter 10 I describe how to use Kotlin constructs to produce more elegant and better-readable application code.

Chapter 12 describes some methods you can use to communicate between components inside your app or between your app and other apps or the outside world.

Chapter 13 handles different devices from a hardware perspective, including smartphones, wearables like smartwatches, Android TV, and Android Auto. Here I also talk about ways to access the camera and sensors and how you can interface with phone calls.

Chapters 14 to 17 deal with testing, troubleshooting, and publishing your app, and Chapter 18 explains how to use the tools provided with the SDK installation (part of Android Studio).

Some Notes About the Code

While in general I try to follow a "clean code" approach for all the code presented in this book, for simplicity I use two anti-patterns you shouldn't follow in your production code.

- I do not use localized string resources. So, whenever you see something like this inside XML resources:

  ```
  android:text = "Some message"
  ```

  what instead you should do is create a string resource and let the attribute refer to it, as shown here:

  ```
  android:text = "@string/message"
  ```

- For logging statements, I always use `LOG` as a tag, as shown here:

  ```
  Log.e("LOG", "The message")
  ```

  In your code, you instead should create a tag like this:

  ```
  companion object {
    val TAG="The class name"
    ...
  }
  ```

  and then use this:

  ```
  Log.e(TAG, "The message")
  ```

Preface

Pro Android with Kotlin is an addition to the popular Apress series for Android development targeting the Java platform. With Kotlin as a highly promising new official language in the Android environment, it allows for more elegant programs compared to the Java standard. This book deals with advanced aspects of a modern Android app. With a thorough description of the important parts of Android system internals and professional-level APIs, advanced user interface topics, advanced development topics, in-depth communication surveys, professional-level hardware topics including looking at devices other than smartphones, a troubleshooting part with guidance on how to fix memory and performance problems, and an introduction to app monetizing, the book is an invaluable resource for developers wanting to build state-of-the-art professional apps for modern Android devices.

This book is not meant to be an introduction to the Kotlin language. For this aim, please take a look at the Kotlin web site or any introductory-level book about Kotlin. What you will find here is an attempt to use as many features of Kotlin to write elegant and stable apps using less code compared to Java.

In 2017, Android versions 8.0 and 8.1 were introduced. In a professional environment, writing apps that depend on new Android 8.*x* features is a bad idea since the worldwide distribution of devices running an 8.*x* version is well below 10 percent as of the writing of this book. But you can write code targeting versions 4.0 all the way up to 8.0 (thus covering almost 100 percent of Android devices) by introducing branches in your code. This is what you will be doing in this book. I still concentrate on modern 8.*x* development, but if I use modern features not available to older versions, I will tell you.

Note that this book does not pay much attention to Android versions older than 4.1 (API level 16). If you look at the online API documentation, you will find a lot of constructs targeting API levels older than 16. Especially when it comes to support libraries, which were introduced to improve backward compatibility, development gets unnecessarily complicated if you look at API versions older than 16 because the distribution of such devices is less than 1 percent nowadays. This book will just assume you are not interested in such old versions, making it unnecessary to look at such support libraries in many cases and simplifying development considerably.

Chapter 1

System

The Android OS was born as the child of the Android Inc. company in 2003 and was later acquired by Google LLC in 2005. The first device running Android came on the market in 2008. Since then it has had numerous updates, with the latest version number at the beginning of 2018 reading 8.1.

Ever since its first build, the market share of the Android OS has been constantly increasing, and by 2018 it is said to be greater than 80 percent. Even though the numbers vary with the sources you use, the success of the Android OS is surely undeniable. This victory partly has its roots in Google LLC being a clever player in the worldwide smartphone market, but it also comes from the Android OS carefully being tailored to match the needs of smartphones and other handheld or handheld-like devices.

The majority of computer developers formerly or still working in the PC environment would do a bad job utterly disregarding handheld device development, and this book's goal is to help you as a developer understand the Android OS and master the development of its programs. The book also concentrates on using Kotlin as a language to achieve development demands, but first we will be looking at the Android OS and auxiliary development-related systems to give you an idea about the inner functioning of Android.

The Android Operating System

Android is based on a specially tailored Linux kernel. This kernel provides all the low-level drivers needed to address the hardware, the program execution environment, and low-level communication channels.

On top of the kernel you will find the *Android Runtime* (ART) and a couple of low-level libraries written in C. The latter serve as a glue between application-related libraries and the kernel. The Android Runtime is the execution engine where Android programs run.

You as a developer hardly ever need to know about the details of how these low-level libraries and the Android Runtime do their work, but you will be using them for basic programming tasks such as addressing the audio subsystem or databases.

P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_1

Above the low-level libraries and the Android Runtime sits the *application framework*, which defines the outer structure of any app you build for Android. It deals with activities, GUI widgets, notifications, resources, and so on. While understanding the low-level libraries certainly helps you to write good programs, knowing the application framework is essential to writing any Android app at all.

On top of all that you will find the apps your users launch for tasks they have to accomplish. See Figure 1-1.

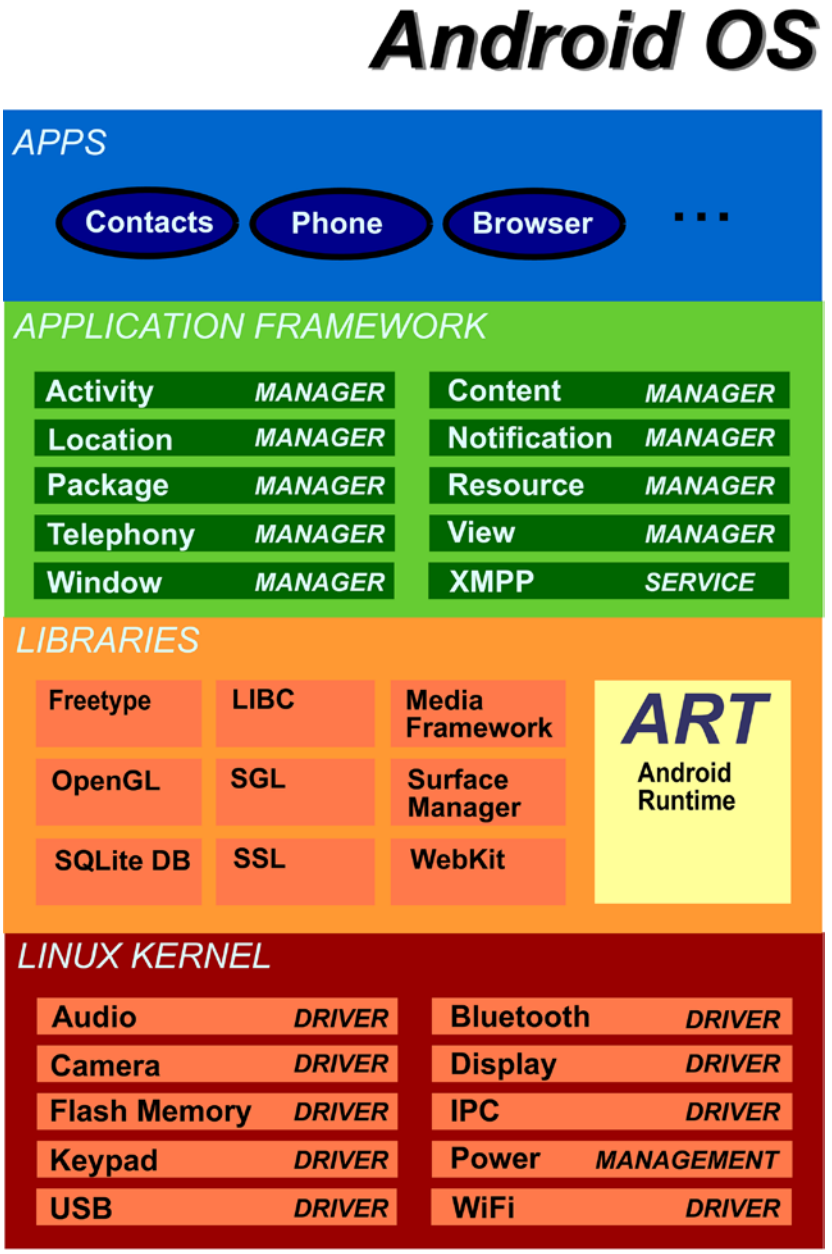

Figure 1-1. *The Android OS*

You as a developer will create Android apps using Kotlin, Java, or C++ as a programming language, or a combination of them. And you will be using the application framework and the libraries to talk to the Android OS and the hardware. Using C++ as a programming language on a lower level, addressing target architecture peculiarities, leads to incorporating the *Native Development Kit* (NDK), which is an optional part of the Android SDK. While for special purposes it might be necessary to use the NDK, in most cases the extra effort to deal with yet another language and the special challenges it bears does not pay off. So in this book, we will be mainly talking about Kotlin, and sometimes Java where appropriate.

The Development System

The operating system running on handhelds is one part of the story; you as a developer also need a system for creating Android apps. The latter happens on a PC or laptop, and the software suite you use for it is *Android Studio*.

Android Studio is the IDE you use for development, but while you install and operate it, the software development kit (see the section "The SDK") gets installed as well, and we will be talking about both in the following sections. We will also cover virtual devices, which provide an invaluable aid for testing your app on various target devices.

Android Studio

The Android Studio IDE is the dedicated development environment for creating and running Android apps. Figure 1-2 shows its main window together with an emulator view.

Android Studio provides the following:

- Managing program sources for Kotlin, Java, and C++ (NDK)
- Managing program resources
- The ability to test-run apps inside emulators or connected real devices
- More testing tools
- A debugging facility
- Performance and memory profilers
- Code inspection
- Tools for building local or publishable apps

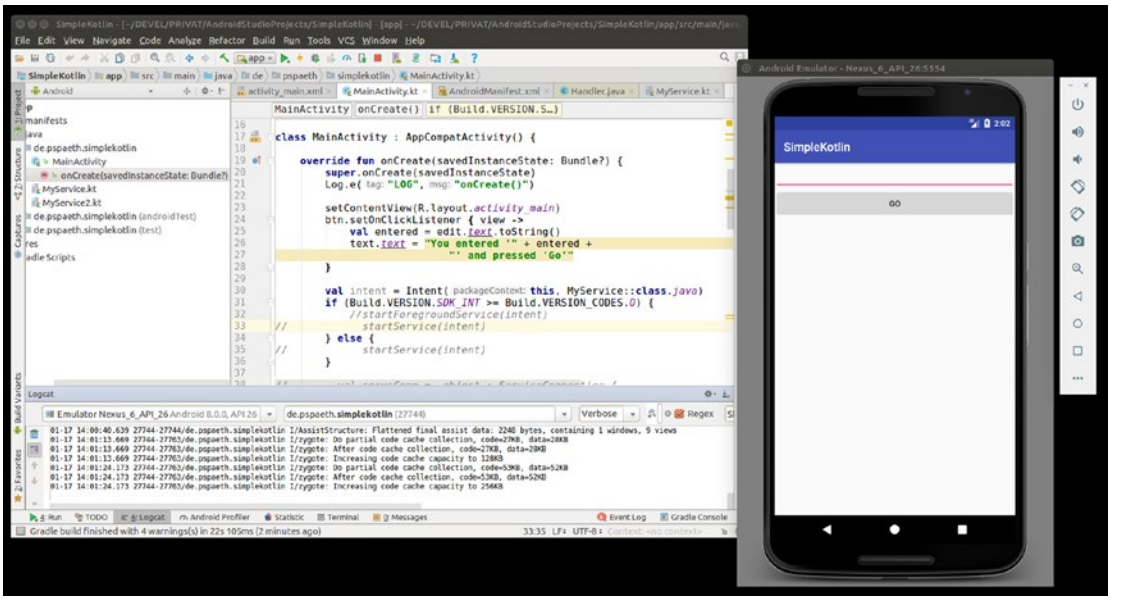

Figure 1-2. Android Studio

The help included in the studio and online resources provide enough information to master Android Studio. In this book, we will be talking about it once in a while and in dedicated chapters.

Virtual Devices

Developing software for computers always included the challenge to create one program that is able to handle all possible target systems. With handheld devices coming in so many different forms nowadays, this aspect has become more critical than ever before. You have smartphone devices with sizes between 3.9” and 5.4” and more, tablets from 7” to 14” and more, wearables, TVs at different sizes, and so on, all running with Android OS.

Of course, you as a developer cannot possibly buy all devices that are needed to cover all possible sizes. This is where emulators come in handy. With emulators you don’t have to buy hardware and you still can develop Android apps.

Android Studio makes it easy for you to use emulators for developing and testing apps, and using the tools from the software development kit you can even operate the emulators from outside Android Studio.

> **Caution** You *can* develop apps without owning a single real device. This is, however, not recommended. You should have at least one smartphone from the previous generation and maybe also a tablet if you can afford it. The reason is that operating real devices feels different compared to emulators. The physical handling is not 100 percent the same, and the performance differs as well.

To manage virtual devices from inside Android Studio, open the Android Virtual Device Manager via *Tools* ➤ *Android* ➤ *AVD Manager*. From here you can investigate, alter, create, delete, and start virtual devices. See Figure 1-3.

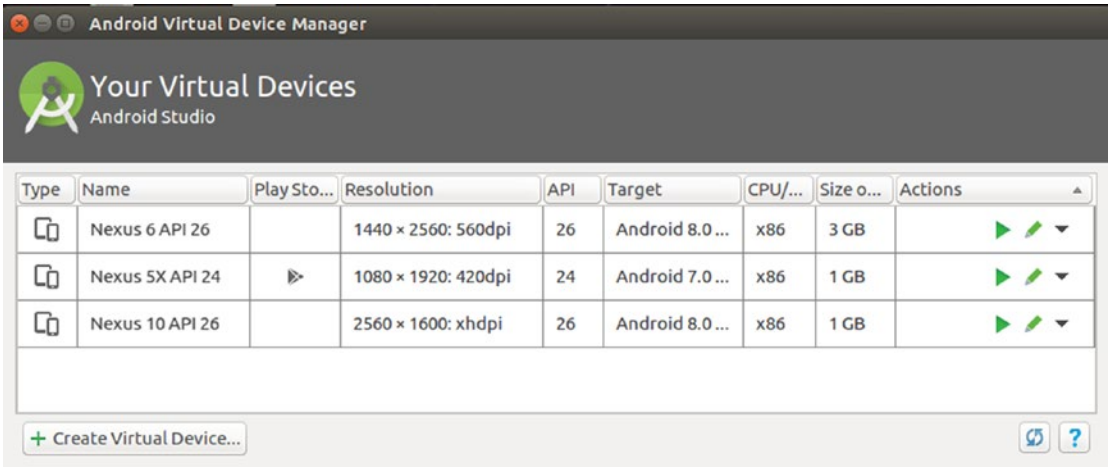

***Figure 1-3** AVD Manager*

When creating a new virtual device, you will be able to choose from a TV, wear, phone, or tablet device; you can select the API level to use (and download new API levels); and in the settings you can specify things like this:

- Graphics performance
- Camera mode (advanced settings)
- Network speed (advanced settings)
- Boot option (advanced settings; quick boot considerably improves bootup speed once the device has been booted for the first time)
- Number of simulated CPUs (advanced settings)
- Memory and storage settings (advanced settings)

The virtual device base images and skins used for creating virtual images can be found here:

```
SDK_INST/system-images
SDK_INST/skins
```

The actual virtual devices with installed apps and user data are in the following location:

```
~/.android/avd
```

Caution Virtual devices do not emulate all hardware supported by real devices. Namely, in the first quarter of 2018, the following are not supported:

- WiFi before API level 25
- Bluetooth
- NFC
- SD card eject and insert
- Headphones attached to the device
- USB

You must thus take precautions inside your app for these not to be present if you want to use the emulator.

Handling running virtual devices can also be done by various command-line tools; see Chapter 18 for more information.

The SDK

The software development kit (SDK) is, in contrast to Android Studio, a loosely coupled selection of tools that are either essential for Android development and as such directly used by Android Studio or at least helpful for a couple of development tasks. They can all be started from within a shell and come with or without their own GUI.

In case you don't know where the SDK was installed during the installation of Android Studio, you can easily ask Android Studio: select *File* ➤ *Project Structure* ➤ *SDK location* from the menu.

The command-line tools that are part of the SDK are described in Chapter 18.

Chapter 2

Application

An Android app consists of components such as *activities*, *services*, *broadcast receivers*, and *content providers*, as shown in Figure 2-1. Activities are for interacting with device users, services are for program parts that run without a dedicated user interface, broadcast receivers listen for standardized messages from other apps and components, and content providers allow other apps and components to access a certain amount and kind of data provided by a component.

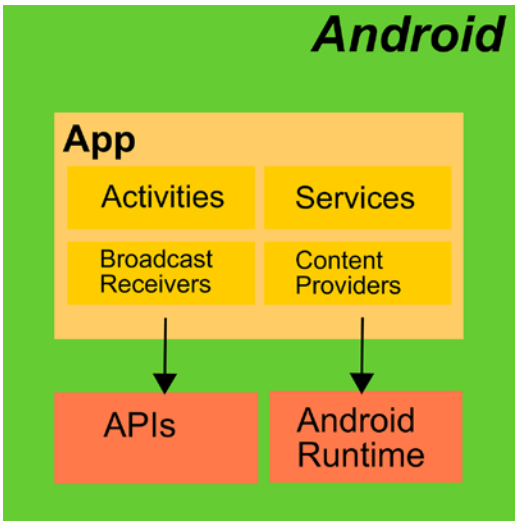

Figure 2-1. *An app in the Android OS*

Components get started by the *Android Runtime*, or execution engine if you like, either by itself or on behalf of other components that create start triggers. When a component gets started depends on its type and the meta-information given to it. At the end of the lifecycle, all running components are subject to removal from the process execution list either because they have finished their work or because the Android OS has decided that

P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_2

a component can be removed because it is no longer needed or that it must be removed because of a device resource shortage.

To make your app or component run as stable as possible and give your users a good feeling about its reliability, a deeper knowledge of the lifecycle of Android components is helpful. We will be looking at system characteristics of components and their lifecycles in this chapter.

Simple apps and Android components are easy to build; just refer to one of the tutorials on the official Android web site or one of the thousand other tutorials elsewhere on the Web. A simple app is not necessarily a professional-level stable app, though, because Android state handling as far as the app is concerned is not the same as for a desktop application. The reason for this is that your Android device might decide to kill your app to save system resources, especially when you temporarily suspend the app in question because you use one or more other apps for some time.

Of course, Android will most likely never kill apps you are currently working with, but you have to take precautions. Any app that was killed by Android can be restarted in a defined data and processing state, including most currently entered data by the user and possibly interfering in the least possible amount with the user's current workflow.

From a file perspective, an Android app is a single zip archive file with the suffix `.apk`. It contains your complete app including all meta-information, which is necessary to run the app on an Android device. The most important control artifact inside is the file `AndroidManifest.xml` describing the application and the components an application consists of.

We do not in detail cover this archive file structure here, since in most cases Android Studio will be taking care of creating the archive correctly for you, so you usually don't need to know about its intrinsic functioning. But you can easily look inside. Just open any `*.apk` file; for example, take a sample app you've already built using Android Studio, as shown here:

```
AndroidStudioProject/[YOUR-APP]/release/app-release.apk
```

***Figure 2-2.** An APK file unzipped*

Then unzip it. APK files are just normal zip files. You might have to temporarily change the suffix to `.zip` so your unzip program can recognize it. Figure 2-2 shows an example of an unzipped APK file.

This `.dex` file contains the compiled classes in *Dalvik Executable* format, something that is similar to a JAR file in Java.

We will be talking about app-related artifacts shortly, but first we will be looking at the more conceptual idea of what tasks are.

Tasks

A *task* is a group of activities interacting with each other in such a way that the end user considers them as the elements of an application. A user starts an app and sees the main activity, does some work there, opens and closes subactivities, maybe switches to another app, comes back, and eventually closes the app.

Going a bit more in-depth, the main structure a task exhibits is its *back stack*, or simply *stack*, where activities of an app pile up. The standard behavior for simple apps in this stack is that the first activity when you launch an app builds the *root* of this stack, the next activity launched from inside the app lands on top of it, another subactivity lands on top of both, and so on. Whenever an activity gets closed because you navigate back (that is where the name *back stack* comes from), the activity gets removed from the stack. When the root activity gets removed, the stack gets closed as a whole, and your app is considered shut down.

Inside the `<application>` element of the `AndroidManifest.xml` file, in more detail described in section “The Application Declaration” of the online text companion, we can see several settings altering the standard behavior of the task stack, and we will see more in Chapter 3. This way, a tailored task stack can become a powerful means to help your end users to understand and fluently use your app. Keep in mind that a complicated stack behavior might be hard to understand for users beginning to use your app, so it should be your aim to find a good balance between power and ease of use.

The Application Manifest

An important central app configuration file you can see in any Android app is the file `AndroidManifest.xml`. It describes the app and declares all the components that are part of the app. The outline of such a manifest file might look like this:

```
<manifest xmlns:android=
              "http://schemas.android.com/apk/res/android"
          xmlns:tools=
              "http://schemas.android.com/tools"
          package="de.pspaeth.tinqly">
   ...
   <application
       android:allowBackup="true"
       android:icon="@mipmap/my_icon"x
       android:label="@string/app_name"
       android:roundIcon="@mipmap/my_round_icon"
       android:supportsRtl="true"
       android:theme="@style/AppTheme">
       <activity ... />
   </application>
</manifest>
```

The most important attribute of the root entry <manifest> is called package. It declares the ID of your app, and if you plan to publish your app, this must be a worldwide unique ID for it. A good idea is to use your domain (or your company's domain) reversed and then an unique application identifier, as shown in the previous code.

Table 2-1 describes all the possible attributes of <manifest>. Note that for the simplest apps, all you need is the package attribute and a single <application> child.

***Table 2-1.** Manifest Main Attributes*

Name	Description
android: installLocation	Defines the installation location. Use internalOnly for installing only in the internal storage, auto for letting the OS decide with affinity toward using the internal storage (the user can switch later in the system settings), or preferExternal for letting the OS decide with affinity toward using the external storage. The default is internalOnly. Note that a couple of restrictions apply to using external storage for that aim; see the online documentation for <manifest>. For modern devices with lots of free internal storage, you should never need to specify preferExternal here.
package	Defines the worldwide unique ID of your app and is a string like abc.def.ghi.[...] where the nondot characters may contain the letters A–Z and a–z, the numbers 0–9, and underscores. Don't use a number after a dot! This is also the default process name and the default task affinity; see the online text companion to learn what those mean. Note that once your app is published, you cannot change this package name in the Google Play Store. There is no default; you must set this attribute.
android: sharedUserId	Defines the name of the Android OS user ID assigned to the app. You can prior to Android 8.0 or API level 26 do things such as assigning the same user ID to different apps, which lets them freely interchange data. The apps must then be signed with the same certificate. However, you normally don't have to set this attribute, but if you set it, make sure you know what you are doing.
android: sharedUserLabel	If you also set sharedUserId, you can set a user-readable label for the shared user ID here. The value must be a reference to a string resource (for example, @string/myUserLabel).
android: targetSandboxVersion	Serves as a security level and is either 1 or 2. Starting with Android 8.0 or API level 26, you *must* set it to 2. With 2, the user ID can no longer be shared between different apps, and the default value for usesClearTextTraffic (see the online text companion) is set to false.
android: versionCode	Defines an internal version number of your app. This is not shown to users and used only for comparing versions. Use an integer number here. This defaults to undefined.
android: versionName	Defines a user-visible version string. This is either the string itself or a pointer to a string resource ("@string/..."). This is not used for anything else but informing the user.

All elements possible as children to the <manifest> element are listed in the section "Manifest Top Level Entries" of the online text companion. The most important one, <application>, describes the application and gets covered in detail in the section "The Application Declaration" of the online text companion.

Chapter 3

Activities

Activities represent user interface entry points of your app. Any app that needs to interact functionally with the user in a direct way, by letting the user enter things or telling the user graphically about the functional state of an app, will expose at least one *activity* to the system. I say *functionally* because telling the user about events can also happen via notifications through *toasts* or the *status bar*, for which an activity is not needed.

Apps can have zero, one, or more activities, and they get started in one of two ways:

- The *main activity*, as declared inside `AndroidManifest.xml`, gets started by launching the app. This is kind of similar to the `main()` function of traditional applications.
- All activities can be configured to be started by an explicit or implicit *intent*, as configured inside `AndroidManifest.xml`. Intents are both objects of a class and a new concept in Android. With explicit intents, by triggering an intent, a component specifies that it needs something to be done by a dedicated component of a dedicated app. For implicit intents, the component just tells what needs to be done without specifying which component is supposed to do it. The Android OS or the user decides which app or component is capable of fulfilling such an implicit request.

From a user perspective, activities show up as things that can be started from inside an application launcher, be it the standard launcher or a specialized third-party launcher app. As soon as they are running, they show up in a task stack as well, and users will see them when using the Back button.

P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_3

Declaring Activities

To declare an activity, you can write the following inside `AndroidManifest.xml`, for example:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...
    package="com.example.myapp">
  <application ... >
      <activity android:name=".ExampleActivity" />
      ...
  </application ... >
  ...
</manifest >
```

As shown in this particular example, you can start the name with a dot, which leads to prepending the app's package name. In this case, the full name of the activity is `com.example.myapp.ExampleActivity`. Or you can write the full name, as shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... package="com.example.myapp" ...>
  <application ... >
      <activity android:name=
          "com.example.myapp.ExampleActivity" />
      ...
  </application ... >
  ...
</manifest>
```

All attributes you can add to the `<activity>` element are listed in the section "Activity Related Manifest Entries" in the online text companion.

The following are elements that can be child elements inside the `activity` element:

- **<intent-filter>**

  This is an intent filter. For details, see the online text companion at "Activity-Related Manifest Entries". You can specify zero, one, or many intent filters.

- **<layout>**

  Starting with Android 7.0, you can specify layout attributes in multiwindow modes as follows, where you of course can use your own numbers:

```
<layout android:defaultHeight="500dp"
        android:defaultWidth="600dp"
        android:gravity="top|end"
        android:minHeight="450dp"
        android:minWidth="300dp" />
```

The attributes `defaultWidth` and `defaultHeight` specify the default dimensions, the attribute `gravity` specifies the initial placement of the activity in freeform modes, and the attributes `minHeight` and `maxHeight` signify minimum dimensions.

- **<meta-data>**

  This is an arbitrary name-value pair in the form `<meta-data android:name="..." android:resource="..." android:value="..." />`. You can have several of them, and they go into an `android.os.Bundle` element available as `PackageItemInfo.metaData`.

> **Caution** Writing an app without any activity is possible. The app can still provide services, broadcast receivers, and data content as a content provider. One thing you as an app developer need to bear in mind is that users do not necessarily understand what such components without user interfaces actually do. In most cases, providing a simple main activity just to give information is recommended and improves the user experience. In a corporate environment, though, providing apps without activities is acceptable.

Starting Activities

Activities can be started in one of two ways. First, if the activity is marked as the launchable main activity of an app, the activity can be started from the app launcher. To declare an activity as a launchable main activity, inside the `AndroidManifest.xml` file you'd write the following:

```
<activity android:name=
      "com.example.myapp.ExampleActivity">
  <intent-filter>
    <action android:name=
          "android.intent.action.MAIN" />
    <category android:name=
          "android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

`android.intent.action.MAIN` tells Android that it is the main activity and will go to the bottom of a task, and the `android.intent.category.LAUNCHER` specifies that it must be listed inside the launcher.

Second, an activity can be started by an intent from the same app or any other app. For this to be possible, inside the manifest you declare an intent filter, as shown here:

```
<activity android:name=
      "com.example.myapp.ExampleActivity">
  <intent-filter>
    <action android:name=
        "com.example.myapp.ExampleActivity.START_ME" />
    <category android:name=
        "android.intent.category.DEFAULT"/>
  </intent-filter>
</activity>
```

The corresponding code to address this intent filter and actually launch the activity now looks like this:

```
val intent = Intent()
intent.action =
    "com.example.myapp.ExampleActivity.START_ME"
startActivity(intent)
```

The flag `exported="false"` must be set for calls from other apps. The category specification `android.intent.category.DEFAULT` inside the filter takes care of the activity being launchable even with no category set in the launching code.

In the previous example, we used an *explicit* intent to call an activity. We precisely told Android which activity to call, and we even expect there to be precisely one activity, which gets addressed this way through its intent filter. The other type of intent is called an *implicit* intent, and what it does, contrary to calling precisely one activity, is tell the system what we actually want to do without specifying *which* app or which component to use. Such implicit calls, for example, look like this:

```
val intent = Intent(Intent.ACTION_SEND)
intent.type = "text/plain"
intent.putExtra(Intent.EXTRA_TEXT, "Give me a Quote")
startActivity(intent)
```

This snippet calls an activity that is able to handle `Intent.ACTION_SEND` actions, receive texts in the MIME type `text/plain`, and pass over the text "Give me a quote." The Android OS will then present the user with a list of activities from this or other apps that are capable of receiving this kind of intent.

Activities can have data associated with them. Just use one of the overloaded `putExtra(...)` methods of the intent class.

Activities and Tasks

What actually happens with a launched activity concerning the task stack gets determined by the attributes, listed here, as given in the `<activity>` element's attributes:

- `taskAffinity`
- `launchMode`
- `allowTaskReparenting`
- `clearTaskOnLaunch`
- `alwaysRetainTaskState`
- `finishOnTaskLaunch`

and by the intent calling flags, listed here:

- `FLAG_ACTIVITY_NEW_TASK`
- `FLAG_ACTIVITY_CLEAR_TOP`
- `FLAG_ACTIVITY_SINGLE_TOP`

You can specify `Intent.flags = Intent.<FLAG>`, where `<FLAG>` is one from the list. In case the activity attributes and caller flags contradict, the caller flags win.

Activities Returning Data

If you start an activity by using this:

```
startActivityForResult(intent:Intent, requestCode:Int)
```

it means you expect the called activity to give something back while it returns. The construct you use in the called activity reads as follows:

```
val intent = Intent()
intent.putExtra(...)
intent.putExtra(...)
setResult(Activity.RESULT_OK, intent)
finish()
```

where inside the `.putExtra(...)` method calls you can add whatever data is to be returned from the activity. You can, for example, add these lines to the `onBackPressed()` event handler method.

For `setResult()`'s first argument, you can use any of the following:

- `Activity.RESULT_OK` if you want to tell the caller the called activity successfully finished its job.
- `Activity.RESULT_CANCELED` if you want to tell the caller the called activity did not successfully finish its job. You still can put extra information via `.putExtra(...)` to specify what went wrong.
- `Activity.RESULT_FIRST_USER + N`, with `N` being any number from 0, 1, 2, ..., for any custom result code you want to define. There is practically no limit for `N` (the maximum value reads $2^{31} - 1$).

Note that you need to take care of also handling back-press events if you have a toolbar. One possibility is to add to the `onCreate()` method lines as follows:

```
setSupportActionBar(toolbar)
supportActionBar!!.setDisplayHomeAsUpEnabled(true)
// The navigation button from the toolbar does not
// do the same as the BACK button, more precisely
// it does not call the onBackPressed() method.
// We add a listener to do it ourselves
toolbar.setNavigationOnClickListener { onBackPressed() }
```

When the called intent returns the way described earlier, the calling component needs to be informed of that event. This is done asynchronously since the `startActivityForResult()` method immediately returns and does not wait for the called activity to finish. The way this event gets caught nevertheless is by overriding the `onActivityResult()` method, as shown here:

```
override
fun onActivityResult(requestCode:Int, resultCode:Int,
      data:Intent) {
   // do something with 'requestCode' and 'resultCode'
   // returned data is inside 'data'
}
```

`requestCode` is whatever you set inside `startActivityForResult()` as `requestCode`, and `resultCode` is what you wrote as the first argument in `setResult()` in the called activity.

> **Caution** On some devices, `requestCode` has its most significant bit set to 1, no matter what was set before. To be on the safe side, you can use the Kotlin construct inside `onActivityResult()` as follows:
>
> ```
> val requestCodeFixed = requestCode and 0xFFFF
> ```

Intent Filters

Intents are objects to tell Android that something needs to be done, and they can be *explicit* by exactly specifying which component needs to be called or *implicit* if we don't specify the called component but let Android decide which app and which component can answer the request. In case there is some ambiguity and Android cannot decide which component to call for implicit intents, Android will ask the user.

For implicit intents to work, a possible intent receiver needs to declare which intents it is able to receive. For example, an activity might be able to show the contents of a text file, and a caller saying "I need an activity that can show me text files" possibly connects to exactly this activity. Now the way the intent receiver declares its ability to answer intent requests is by specifying one or more *intent filters* in its app's `AndroidManifest.xml` file. The syntax of such a declaration is as follows:

```
<intent-filter android:icon="drawable resource"
               android:label="string resource"
               android:priority="integer" >
    ...
</intent-filter>
```

Here, `icon` points to a drawable resource ID for an icon, and `label` points to a string resource ID for a label. If unspecified, the icon or label from the parent element will be used. The `priority` attribute is a number between -999 and 999 and for intents specifies its ability to handle such intent request, and for receivers specifies the execution order for several receivers. Higher priorities come before lower priorities.

> **Caution** The `priority` attribute should be used with caution. A component cannot possibly know what priorities other components from other apps can have. So, you introduce some kind of dependency between apps, which is not intended by design.

This `<intent-filter>` element can be a child of the following:

- `<activity>` and `<activity-alias>`
- `<service>`
- `<receiver>`

So, intents can be used to launch activities and services and to fire broadcast messages.

The element must contain children elements as follows:

- `<action>` (obligatory)
- `<category>` (optional)
- `<data>` (optional)

Intent Action

The `<action>` child of the filter (or children, because you can have more than one) specifies the action to perform. The syntax is as follows:

```
<action android:name="string" />
```

This will be something expressing an action such as View, Pick, Edit, Dial, and so on. The complete list of generic actions is specified by constants with names like `ACTION_*` inside the class `android.content.Intent`; you can find a list in the section "Intent Constituent Parts" in the online text companion. Besides those generic actions, you can define your own actions.

> **Note** Using any of the standard actions does not necessarily mean there is any app on your device that is able to respond to a corresponding intent.

Intent Category

The `<category>` child of the filter specifies a category for the filter. The syntax is as follows:

```
<category android:name="string" />
```

This attribute may be used to specify the type of component that an intent should address. You can specify several categories, but the category is not used for all intents, and you can omit it as well. The filter will match the intent only if *all* required categories are present.

When an intent is used on the invoker side, you can add categories by writing the following, for example:

```
val intent:Intent = Intent(...)
intent.addCategory("android.intent.category.ALTERNATIVE")
```

Standard categories correspond to constants with names like `CATEGORY_*` inside the `android.content.Intent` class. You can find them listed in the section "Intent Constituent Parts" in the online text companion.

> **Caution** For implicit intents, you *must* use the `DEFAULT` category inside the filter. This is because the methods `startActivity()` and `startActivityForResult()` use this category by default.

Intent Data

The `<data>` child of the filter is a data type specification for the filter. The syntax is as follows:

```
<data android:scheme="string"
      android:host="string"
      android:port="string"
      android:path="string"
      android:pathPattern="string"
      android:pathPrefix="string"
      android:mimeType="string" />
```

You can specify either of the following or both of the following:

- A data type specified by only the `mimeType` element, for example, `text/plain` or `text/html`. So, you can write the following:

```
<data android:mimeType="text/html" />
```

- A data type specified by scheme, host, port, and some path specification: `<scheme>://<host>:<port>[<path>|<pathPrefix>|<pathPattern>]`. Here `<path>` means the full path, `<pathPrefix>` is the start of a path, and `<pathPattern>` is like a path but with wildcards: `X*` is zero or more occurrences of the character *X*, and `.*` is zero or more occurrences of any character. Because of escaping rules, write `\\*` for an asterisk and `\\\\` for a backslash.

On the caller side, you can use `setType()`, `setData()`, and `setDataAndType()` to set any data type combination.

Caution For implicit intent filters, if the caller specifies a URI `data` part as in `intent.data = <some URI>`, it might not be sufficient to specify just the scheme/host/port/path inside the filter declaration. Under these circumstances, you also have to specify the MIME type, as in `mimeType="*/*"`. Otherwise, the filter possibly won't match. This generally happens in a *content provider* environment since the content provider's `getType()` method gets called for the specified URI and the result gets set as the intent's MIME type.

Intent Extra Data

Any intent can have extra data added to it that you can use to send data with it other than specified by the `<data>` subelement.

While you can use one of the various `putExtra(...)` methods to add any kind of extra data, there are also a couple of standard extra data strings sent by `putExtra(String,Bundle)`. You can find the keys in the section "Intent Constituent Parts" in the online text companion.

Intent Flags

You can set special intent handling flags by invoking the following:

```
intent.flags = Intent.<FLAG1> or Intent.<FLAG2> or ...
```

Most of these flags specify how the intent gets handled by the Android OS. Specifically, flags of the form `FLAG_ACTIVITY_*` are aimed at activities called by `Context.startActivity(..)`, and flags like `FLAG_RECEIVER_*` are for use with `Context.sendBroadCast(...)`. The tables in the section "Intent Constituent Parts" in the online text companion show the details.

System Intent Filters

The system apps (that is, the apps already installed when you buy a smartphone) have intent filters you can use to call them from your app. Unfortunately, it is not that easy to guess how to call the activities from system apps, and relevant documentation is hard to find. A way out is to extract this information from their APK files. This is done for you for API level 26, and the result is listed in the online text companion in the section "The System Intent Filters."

As an example, suppose you want to send an e-mail. Looking at the system intent table in the online text companion, you can find a lot of actions for `PrebuiltGmail`. Which one do we use? Well, first a general-purpose interface should not have too many input parameters. Second, we can also look at the action name to find something that seems appropriate. A promising candidate is the `SEND_TO` action; all that it apparently needs is a `mailto:` data specification. And as it happens, this is the action we actually need. Using an elaborated `mailto:...` URL allows us to specify more recipients, CC and BCC recipients, a subject, and even the mail body. However, you can also just use `"mailto:master@universe.com"` and add recipients, body, and so on, by using extra fields. So to send an e-mail, while possibly letting the user choose among several e-mail apps installed on a device, write the following:

```
val emailIntent:Intent = Intent(Intent.ACTION_SENDTO,
    Uri.fromParts("mailto","abc@gmail.com", null))
```

```
emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Subject")
emailIntent.putExtra(Intent.EXTRA_TEXT, "Body")
startActivity(Intent.createChooser(
    emailIntent, "Send email..."))
// or startActivity(emailIntent) if you want to use
// the standard chooser (or none, if there is only
// one possible receiver).
```

> **Caution** It is at the receiving app's discretion how to exactly handle intent URIs and extra data. A poorly designed e-mailer might not allow you to specify e-mail header data at all. To be on the safe side, you may want to add all header data in both the `mailto:` URI *and* as extra data.

Activities Lifecycle

Activities have a lifecycle, and contrary to traditional desktop applications, they are intentionally subject to being killed whenever the Android OS decides to do so. So, you as a developer need to take special precautions to make an app stable. More precisely, an activity finds itself in one of the following states:

- *Shut down*: The activity is not visible and not processing anything. Still, the app containing the activity might be alive because it has some other components running.
- *Created*: Either the activity is the main activity and was started by the user or some other component or it is an activity regardless of whether it is main activity and it was started by some other component, from inside the same app or another app if security considerations permit it. Also, activity creation happens when you, for example, flip the screen and the app needs to be built up with different screen characteristics. During the creation process, the callback method `onCreate()` gets called. You must implement this method since there the GUI needs to be built up. You can also use this callback method to start or connect to services or provide content provider data. And you can use the APIs to *prepare* playing music, operating the camera, or doing anything else the app is made for. This is also a good place to initially set up a database or other data storage your app needs.
- *Started*: Once done with the creation (and also in case of a restart after a stop), the activity goes into the *started* state. Here the activity is about to become visible to the user. During the start process, the callback method `onStart()` gets called. This is a good place to start broadcast receivers, start services, and rebuild internal state and processes you quit while the activity went to the stopped state.
- *Resumed*: Shortly before actually becoming visible to the user, the activity goes through the resuming process. During that process the callback `onResume()` gets called.
- *Running*: The activity is fully visible, and the user can interact with it. This state immediately follows the resuming process.

- *Paused*: The activity loses focus but is still at least partly visible. Losing the focus, for example, happens when the user taps the Back or Recents button. The activity may continue to send updates to the UI or continue to produce sound, but in the majority of cases the activity will proceed to the stopped state. During the pausing, the `onPause()` callback gets called. The paused state is followed by the stopped state or the resumed state.
- *Stopped*: The activity is invisible to the user. It later might be restarted, destroyed, and expunged from the active process list. During stopping, the `onStop()` callback gets called. After stopping, either destruction or starting happens. Here you can, for example, stop the service you started in `onStart()`.
- *Destroyed*: The activity is removed. The callback `onDestroy()` gets called, and you should implement it and do everything there to release resources and do other cleanup actions.

Table 3-1 lists the possible transitions between an activity's states, which are illustrated in Figure 3-1.

***Table 3-1.** Activity State Transitions*

From	To	Description	Implement
Shut Down	Created	An activity gets called the first time or after a destruction.	`onCreate()`: Call `super.onCreate()`, prepare the UI, start services.
Created	Started	An activity starts after creation.	`onStart()`: You can start services here that are needed only while the activity is visible.
Started	Resumed	The resumed state automatically follows a started state.	Use `onResume`.
Resumed	Running	The running state automatically follows a resumed state.	The activity's functioning including UI activity happens here.
Running	Paused	The activity loses focus because the user tapped the Back or Recents button.	Use `onPause`.
Paused	Resumed	The activity has not stopped yet, and the user navigates back to the activity.	Use `onResume()`.
Paused	Stopped	The activity is invisible to the user, for example, because another activity gets started.	`onStop()`: You can stop services here that are needed only while the activity is visible.
Stopped	Started	A stopped activity gets started again.	`onStart()`: You can start services here that are needed only while the activity is visible.
Stopped	Destroyed	A stopped activity gets removed.	`onDestroy()`: Release all resources, do a cleanup, and stop services that were started in `onCreate`.

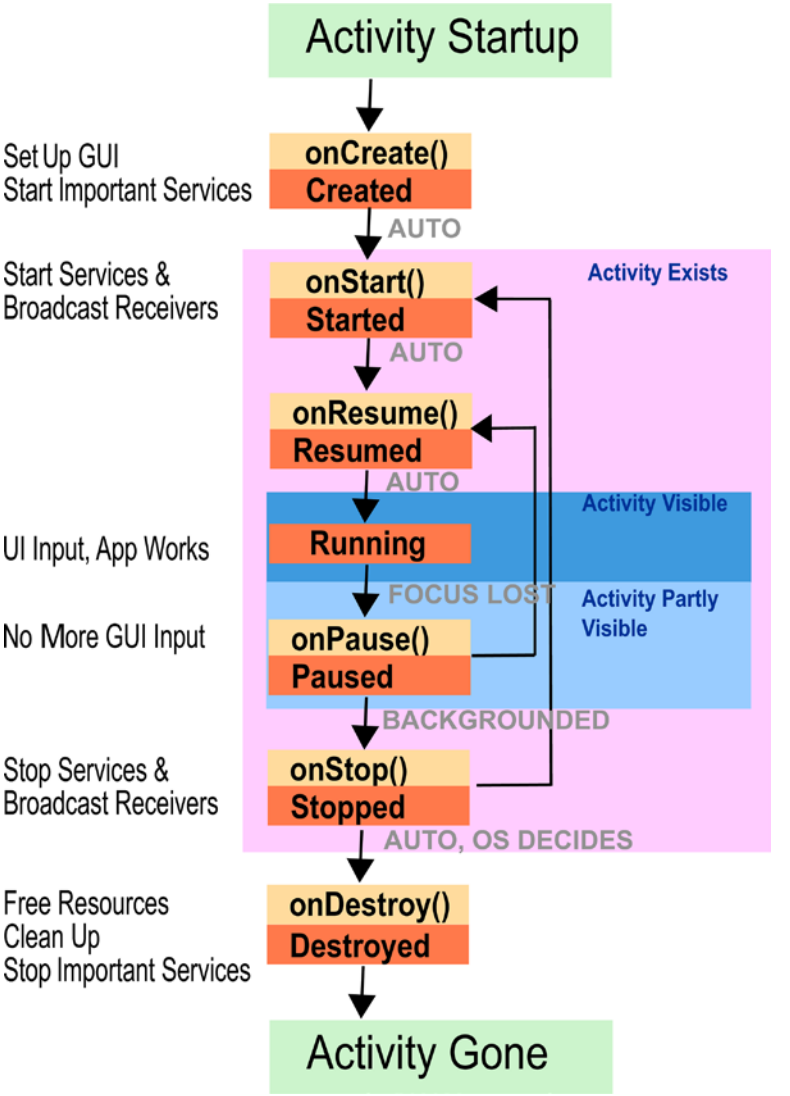

Figure 3-1. Activity state transitions

Preserving State in Activities

I have stressed the need for you to take precautions to make sure your app restarts in a well-behaving manner when it forcibly gets stopped by the Android OS. Here I give you some advice how that can be done.

Looking at the lifecycle of an activity, we can see that an activity about to be killed by the Android OS calls the method `onStop()`. But there are two more callbacks we haven't talked about yet. They have the names `onSaveInstanceState()` and `onRestoreInstanceState()`, and they get called whenever Android decides that an activity's data need to be saved or restored. This is not the same as `onStart()` and `onStop()` because it is sometimes not

necessary to preserve the state of an app. For example, if an activity will not be destroyed but just suspended, the state is going to be kept anyway, and `onSaveInstanceState()` and `onRestoreInstanceState()` will not be invoked.

Android helps us here: the default implementations of `onSaveInstanceState()` and `onRestoreInstanceState()` already save and restore UI elements that have an ID. So, if that is all you need, you don't have to do anything. Of course, your activity might be more complex and may contain other fields you need to preserve. In this case, you can override both `onSaveInstanceState()` and `onRestoreInstanceState()`. Just make sure you call the superclass's methods; otherwise, you must take care of all UI elements yourself.

```
override
fun onSaveInstanceState(outState:Bundle?) {
    super.onSaveInstanceState(outState)
    // add your own data to the Bundle here...
    // you can use one of the put* methods here
    // or write your own Parcelable types
}

override
fun onRestoreInstanceState(savedInstanceState: Bundle?) {
    super.onRestoreInstanceState(savedInstanceState)
    // restore your own data from the Bundle here...
    // use one of the get* methods here
}
```

Note that the saved state goes to the `onCreate()` callback as well, so it is up to you whether you want to use the `onRestoreInstanceState()` or `onCreate()` method for restoring the state.

Under these circumstances, the standard mechanism for saving and restoring state might not suit your needs. For example, it can't preserve data when you stop your app. In this case, the `onSaveInstanceState()` method is not getting called. If you need to preserve data in such cases, you can use `onDestroy()` to save your app's data in a database and read the database during the `onCreate()` callback. See Chapter 8 for more information.

Chapter 4

Services

Services are components running without a user interface and with a conceptual affinity toward long-running processes. They are separate from notifications in the *status bar* or a *Toast*. Services can be started by apps, or they can be bound to by apps, or both.

Services come in two flavors: foreground services and background services. While at first glance it seems to be a contradiction to speak of "foreground" services since so many people tend to say that "services run in the background," foreground services do actually exist. The distinction between foreground and background services is crucial because their behaviors are different.

> **Caution** Do not misinterpret services as constructs for running anything that needs to be calculated in the background, in other words, not disturbing GUI activities. If you need a process that does not interfere with the GUI but is otherwise not eligible to run while your app is inactive and also not subject to being used from outside your app, consider using a thread instead. See Chapter 10 for more information.

P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_4

Foreground Services

The intrinsic functioning of foreground services differs with different Android versions. While foreground services prior to Android 8.0 (API level 26) were just background services with an entry inside the status bar and otherwise no stringent influence on how the Android OS handles them, with Android 8.0 (API level 26), foreground services follow a special notation and receive improved attention from the Android OS, making them less likely to be killed because of resource shortages. Here are some details:

- **Foreground services before Android 8.0 (API level 26)** are services that just present a notification entry in the status bar. The client component that needs to use a service doesn't know whether the service that started is a foreground service or not; it just starts the service via `startService(intent)`. See Chapter 12.
- **Foreground services starting with Android 8.0 (API level 26)** run with the user being made aware of them. They *must* interfere with the operating system by notifications in the status bar. A client component explicitly starts a foreground service by invoking `startForeroundService(intent)`, and the service itself must readily tell the Android OS within a few seconds that it wants to run as a foreground service by calling `startForeground(notificationId, notification)`.

One noticeable lifetime characteristics of a foreground service is it is less likely to be killed by Android because of an available resource shortage. The documentation is, however, not precise about that. Sometimes you'll read "will not be killed" and sometimes "less likely to be killed." Also, the way Android handles such things is subject to change with new Android versions. As a general rule of thumb, you should be conservative and expect the worst. In this case, read "less likely to be killed" and take precautions if the service ceases functioning while your app is performing some work.

Background Services

Background services run in the background; that is, they will not show an entry in the status bar. They are, however, allowed to use *Toasts* to send short notification messages to the user. Background services are more brittle compared to foreground services since Android expects them to be more loosely connected to user activities and thus more readily decides to kill them when there is a resource shortage.

Starting with Android 8.0 (API level 26), a couple of limitations hold if you are instantiating background services the old way, and a shift toward using the *JobScheduler* methodology is recommended. Apps running on Android 8.0 or newer are considered to run in the background, if none of the following is true:

- The app has a visible activity, currently active or paused.
- The app has a foreground service, in other words, a service has called `startForegound()` during its operation.
- Another foreground app is connected to it, either by using one of its services or by using it as a content provider.

Once an Android 8.0 app starts its life as a background app or is switched to being a background app, it has a window of a couple of minutes before it is considered *idle*. Once idle, the background services of an app get stopped. As an exception to this, a background app will go on a *whitelist* and is allowed to execute background services if it handles tasks visible to the user. Examples include handling a "Firebase Cloud Messaging" message, receiving a broadcast such as an SMS or MMS message, executing a `PendingIntent` from a notification (an intent to be executed by a different app with the originating app's permission), or starting a `VpnService`.

Most things that were formerly accomplished by executing background jobs as of Android 8.0 are considered to be eligible to be handled by the JobScheduler API; see Chapter 8 for more information.

Declaring Services

Services get declared inside the app's `AndroidManifest.xml` file as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
    <application ...>
        <activity ...>
        </activity>
        <service
             android:name=".MyService"
             android:enabled="true"
             android:exported="true">
        </service>
    </application>
</manifest>
```

See Table 4-1 for the flags available.

Table 4-1. Manifest Flags for Services

Name	Description
`android:description`	This is a resource ID pointing to a description of the service. You should use it because users can kill services, but if you tell them what your service does, this is less likely to happen.
`android:directBootAware`	This can be `true` or `false`. The default is `false`. If `true`, the service can run even if the device is not yet unlocked after a restart. The Direct Boot mode was introduced in Android 7.0 (API level 24). Note that a Direct Boot–aware service must store its data in the device's protected storage.
`android:enabled`	This can be `true` or `false`. The default is `true`. If `false`, the service is effectively disabled. You wouldn't normally set it to `false` for a production service.
`android:exported`	This can be `true` or `false`. This specifies whether other applications can use the service. The default is `false` if there are no intent filters and `true` otherwise. The presence of intent filters implies external usage, thus this distinction.
`android:icon`	This is the icon resource ID. The default is the app's icon.
`android:isolatedProcess`	This can be `true` or `false`. The default is `false`. If `true`, the service has no means to communicate with the system, only through the service methods. It is actually a good idea to use this flag, but in most cases your service will need to talk to the system, so you have to leave it `false` unless the service is really self-contained.
`android:label`	This is a label for the service displayed to the user. The default is the app's label.
`android:name`	This is the name of the service's class. If you use a dot as the first character, it automatically gets prepended with the name of the package specified in the `manifest` element.
`android:permission`	This is the permission name adjoint to this service. The default is the `permission` attribute in the `application` element. If not specified and a default does not apply, the service will not be protected.
`android:service`	This is the name of the service's process. If specified, the service will run in its own process. If it starts with a colon (`:`), the process will be private to the app. If it starts with a lowercase letter, the process spawned will be a global process. Security restrictions might apply.

The `<service>` element allows for the following child elements:

- **<intent-filter>**

  This can be zero, one, or many intent filters. They are described in Chapter 3.

- **<meta-data>**

  This is an arbitrary name-value pair in the form `<meta-data android:name="..." android:resource="..." android:value="..." />`. You can have several of them, and they go into an `android.os.Bundle` element available as `PackageItemInfo.metaData`.

For you as a professional developer, understanding what a *process* actually is and how it gets treated by the Android OS is quite important; see the `android:service` flag in the manifest for process control. It can be tricky because process internals tend to change with new Android versions, and they seem to change on a minute-by-minute basis if you read blogs. As a matter of fact, a *process* is a computational unit that gets started by the Android OS to perform computational tasks. Also, it gets stopped when Android decides it has run out of system resources. If you decide to stop working with a particular app, it doesn't automatically mean the corresponding process or processes get killed. Whenever you start an app for the first time and you don't explicitly tell the app to use another app's process, a new process gets created and started, and with subsequent computational tasks existing, processes get used or new processes get started, depending on their settings and relations to each other.

Unless you explicitly specify service characteristics in the manifest file, a service started by an app will run in the app's process. This means the services possibly live and inevitably die with the app. A process needs to be started to actually live, but when it runs in the app's main process, it will die when the app dies. This means a service's resources needs matter to the app's resources needs. In former times when resources were scarcer, this was more important than nowadays with stronger devices, but it is still good to know about. If a service needs a lot of resources and there is a shortage of resources, it makes a difference if the whole app or just that greedy service needs to be killed to free resources.

If you, however, tell the service to use its own process by virtue of the `android:service` manifest entry, the service's lifecycle can be treated independently by the Android OS. You have to decide: either let it use its own process and accept a possible proliferation of processes for just one app or let them run in one process and couple the lifecycles more closely.

Letting several computation units run in one process has another consequence: they do not run concurrently! This is crucial for GUI activities and processes because we know GUI activities must be fast to not obstruct user interactions, and services are conceptionally bound to longer-running computations. A way out of this dilemma is to use asynchronous tasks or threads. Chapter 10 will be talking more about concurrency.

If the service needs to address the *device protected storage*, as in the Direct Boot mode triggered by the `android:directBootAware` flag in the manifest, it needs to access a special context.

```
val directBootContext:Context =
      appContext.createDeviceProtectedStorageContext()
// For example open a file from there:
val inStream:FileInputStream =
      directBootContext.openFileInput(filename)
```

You should not use this context normally, only for special services that need to be active directly after the boot process.

Service Classes

Services must extend the following class or one of its subclasses:

```
android.app.Service
```

They must be declared inside the app's `AndroidManifest.xml` file, as described earlier.

The interface methods from `android.app.Service` are described in the section "Intent Constituent Parts" in the online text companion.

Note that there are two ways to stop a service that was explicitly started via `startService()` or `startForeroundService`: the service stops itself by calling `stopSelf()` or `stopSelfResult()` or by calling `stopService()` from outside.

Starting Services

A service can be explicitly started from any component that is a subclass of `android.content.Context` or has access to a `Context`. This is the case for activities, other services, broadcast receivers, and content providers.

To explicitly start a service, you need an appropriate intent. We basically have two cases: first, if the service lives in the same app as the client (invoker) of the service, you can write the following for a foreground service as defined starting at Android 8.0 (API level 26):

```
val intent = Intent(this, TheService::class.java)
startService(intent)
```

for a normal service, or

```
val intent = Intent(this, TheService::class.java)
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    startForegroundService(intent)
} else {
    startService(intent)
}
```

So, we can directly refer to the service class. The `TheService::class.java` notation might look strange at first glance if you are a new Kotlin developer; that is just the Kotlin way of providing Java classes as an argument. (For versions prior to Android 8.0 (API level 26), you start it the normal way.)

> **Note** Since intents allow general-purpose extra attributes by using one of the various `putExtra()` methods, we can also pass data to the service.

The second case is given if the service we want to start is part of another app and thus is an *external* service. You then have to add an intent filter inside the service declaration. Here's an example:

```
<service
    android:name=".MyService"
    android:enabled="true"
    android:exported="true">
    <intent-filter>
       <action android:name="<PCKG_NAME>.START_SERVICE" />
    </intent-filter>
</service>
```

In this example, `<PCKG_NAME>` is the name of app's package, and instead of `START_SERVICE`, you can write a different identifier if you like. Now, inside the service client, you can write the following to start and stop the external service, where inside the intent constructor you have to write the same string as in the intent filter declaration of the service:

```
val intent = Intent("<PCKG_NAME>.START_SERVICE")
intent.setPackage("<PCKG_NAME>")
startService(intent)

// ... do something ...

stopService(intent)
```

The `setPackage()` statement is important here (of course you have to substitute the service's package name); otherwise, a security restriction applies, and you get an error message.

Binding to Services

Starting a service is one part of the story. The other part is using them while they are running. This is what the *binding* of services is used for.

To create a service that can be bound to or from the same app, write something like this:

```
/**
 * Class used for binding locally, i.e. in the same App.
 */
class MyBinder(val servc:MyService) : Binder() {
    fun getService():MyService {
        return servc
    }
}
```

```
class MyService : Service() {
    // Binder given to clients
    private val binder: IBinder = MyBinder(this)

    // Random number generator
    private val generator: Random = Random()

    override
    fun onBind(intent: Intent):IBinder  {
        return binder
    }

    /** method for clients */
    fun getRandomNumber():Int {
        return generator.nextInt(100)
    }
}
```

To bind to this service internally, from the same app, inside the service using the client, write the following:

```
val servcConn = object : ServiceConnection {
    override
    fun onServiceDisconnected(compName: ComponentName?) {
        Log.e("LOG","onServiceDisconnected: " + compName)
    }
    override
    fun onServiceConnected(compName: ComponentName?,
                           binder: IBinder?) {
        Log.e("LOG","onServiceConnected: " + compName)
        val servc = (binder as MyBinder).getService()
        Log.i("LOG", "Next random number from service: " +
             servc.getRandomNumber())
    }
    override
    fun onBindingDied(compName:ComponentName) {
        Log.e("LOG","onBindingDied: " + compName)
    }
}
val intent = Intent(this, MyService::class.java)
val flags = BIND_AUTO_CREATE
bindService(intent, servcConn, flags)
```

Here, the `object: ServiceConnection {...}` construct is the Kotlin way of implementing an interface by creating an object of an anonymous inner class, like `new ServiceConnection() {...}` in Java. The construct is called an *object expression* in Kotlin. The `this` inside the intent constructor in this case refers to a `Context` object. You can use it like this inside an activity. If you have the `Context` in a variable instead, use that variable's name here.

Of course, instead of the logging, you should do more meaningful things. Especially inside the `onSeviceConnected()` method you can save the binder or service in a variable for further use. Just make sure, having said all that, that you appropriately react to a died binding or a killed service connection. You could, for example, try to bind the service again, tell the user, or both.

The previous code starts the service automatically once you bind to it and it doesn't exist yet. This happens by virtue of this statement:

```
val flags = BIND_AUTO_CREATE
[...]
```

If you don't need it because you are sure the service is running, you can omit it. In most cases, it is however better to include that flag. The following are the other flags you can use for setting binding characteristics:

- `BIND_AUTO_CREATE`: We just used that; it means the service gets started automatically if it hasn't started yet. You'll sometimes read that explicitly starting a service is unnecessary if you bind to it, but this is true only if you set this flag.
- `BIND_DEBUG_UNBIND`: This leads to saving the callstack of a following `unbindService()`, just in case subsequent unbind commands are wrong. If this happens, a more verbose diagnostic output will be shown. Since this imposes a memory leak, this feature should be used only for debugging purposes.
- `BIND_NOT_FOREGROUND`: This is applicable only if the client runs in a foreground process and the target service runs in a background process. With this flag, the binding process will not raise the service to a foreground scheduling priority.
- `BIND_ABOVE_CLIENT`: With this flag, we specify that the service is more important than the client (i.e., service invoker). In case of a resource shortage, the system will kill the client prior to the service invoked.
- `BIND_ALLOW_OOM_MANAGEMENT`: This flags tell the Android OS that you are willing to accept Android treating the binding as noncritical and killing the service under low memory circumstances.
- `BIND_WAIVE_PRIORITY`: This flags leads to leaving the scheduling of the service invocation up to the process where the service runs in.

Just add them in a combination that suits your needs.

> **Note** Binding is not possible from inside a `BroadcastReceiver` component, unless the `BroadcastReceiver` has been registered via `registerReceiver(receiver.intentfilter)`. In the latter case, the lifetime of the receiver is tied to the registering component. You can, however, from broadcast receivers pass instruction strings inside the intent you used for starting (in other words, not binding) the service.

To bind to an external service, in other words, a service belonging to another app, you cannot use the same binding technique as described for internal services. The reason for this is the `IBinder` interface we are using cannot directly access the service class since the class is not visible across process boundaries. We can, however, wrap data to be transported between the service and the service client into an `android.os.Handler` object and use this object to send data from the service client to the service. To achieve this, for the service we first need to define a `Handler` for receiving messages. Here's an example:

```
internal class InHandler(val ctx: Context) : Handler() {
    override
    fun handleMessage(msg: Message) {
        val s = msg.data.getString("MyString")
        Toast.makeText(ctx, s, Toast.LENGTH_SHORT).show()
    }
}
[...]
class MyService : Service() {
    val myMessg:Messenger = Messenger(InHandler(this))
    [...]
}
```

Instead of just creating a `Toast` message, you can of course do more interesting things when a message arrives. Now in the service's `onBind()` method, we return the binder object provided by the messenger.

```
override
fun onBind(intent:Intent):IBinder {
    return myMessg.binder
}
```

As for the entries inside the `AndroidManifest.xml` file, we can write the same as when *starting* remote services.

In the service client, you'd add a `Messenger` attribute and a `ServiceConnection` object. Here's an example:

```
var remoteSrvc:Messenger? = null
private val myConnection = object : ServiceConnection {
    override
    fun onServiceConnected(className: ComponentName,
                           service: IBinder) {
        remoteSrvc = Messenger(service)
    }
    override
    fun onServiceDisconnected(className: ComponentName) {
        remoteSrvc = null
    }
}
```

To actually perform the binding, you can proceed like for internal services. For example, inside an activity's onCreate() method, you could write the following:

```
val intent:Intent = Intent("<PCKG_NAME>.START_SERVICE")
intent.setPackage("<PCKG_NAME>")
bindService(intent, myConnection, Context.BIND_AUTO_CREATE)
```

Here, substitute the service package's name for <PCKG_NAME> appropriately.

Now to send a message from the client to the service across the process boundary, you can write the following:

```
val msg = Message.obtain()
val bundle = Bundle()
bundle.putString("MyString", "A message to be sent")
msg.data = bundle
remoteSrvc?.send(msg)
```

Note that for this example you cannot add these lines into the activity's onCreate() method after the bindService() statement because remoteSrvc gets a value only after the connection fires up. But you could, for example, add it to the onServiceConnected() method of the ServiceConnection class.

> **Note** In the previous code, no precautions were taken to ensure connection sanity. You should add sanity checks for productive code. Also, unbind from the service inside the onStop() method.

Data Sent by Services

Up to now we were talking of messages sent from the service client to the service. Sending data in the opposite direction, from the service to the service client, is possible as well; it can best be achieved by using an extra Messenger inside the client, a broadcast message, or a ResultReceiver class.

For the first method, provide another Handler and Messenger in the service client, and once the client receives an onServiceConnected() callback, send a Message to the service with the second Messenger passed by the replyTo parameter.

```
internal class InHandler(val ctx: Context) : Handler() {
    override
    fun handleMessage(msg: Message) {
        // do something with the message from the service
    }
}

class MainActivity : AppCompatActivity() {
    private var remoteSrvc:Messenger? = null
    private var backData:Messenger? = null
```

```
    private val myConn = object : ServiceConnection {
      override
      fun onServiceConnected(className: ComponentName,
              service: IBinder) {
          remoteSrvc = Messenger(service)
           backData = Messenger(
                InHandler(this@MainActivity))

           // establish backchannel
           val msg0 = Message.obtain()
           msg0.replyTo = backData
           remoteSrvc?.send(msg0)

           // handle forward (client -> service)
           // connectivity...
    }

    override
    fun onServiceDisconnected(clazz: ComponentName) {
         remoteSrvc = null
    }
}

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

        // bind to the service, use ID from the manifest!
        val intent = Intent("<PCKG>.START_SERVICE")
        intent.setPackage("<PCKG>")
        val flags = Context.BIND_AUTO_CREATE
        bindService(intent, myConn, flags)
    }
}
```

The service can then use this message, extract the `replyTo` attribute, and use it to send messages to the service client.

```
internal class IncomingHandler(val ctx: Context) :
         Handler() {
    override
    fun handleMessage(msg: Message) {
        val s = msg.data.getString("MyString")
        val repl = msg.replyTo
        Toast.makeText(ctx, s, Toast.LENGTH_SHORT).show()
        Log.e("IncomingHandler", "!!! " + s)
        Log.e("IncomingHandler", "!!! replyTo = " + repl)

        // If not null, we can now use the 'repl' to send
        // messages to the client. Of course we can save
        // it elsewhere and use it later as well
        if(repl != null) {
          val thr = Thread( object : Runnable {
```

```
            override fun run() {
                Thread.sleep(3000)
                val msg = Message.obtain()
                val bundle = Bundle()
                bundle.putString("MyString",
                      "A reply message to be sent")
                 msg.data = bundle
                repl?.send(msg)
            }
          } )
          thr.start()
        }
    }
}
```

The other two methods, using a broadcast message or a `ResultReceiver` class, get handled in Chapters 5 and 12.

Service Subclasses

Up to now we were always using `android.app.Service` as a base class for services we described. There are other classes supplied by Android that are usable as base classes, though, with different semantics. For Android 8.0, there are no less than 20 service classes or base classes you can use. You can see them all in the Android API documentation in the "Known Direct Subclasses" section.

> **Note** At the time of writing this book, you can find this documentation at `https://developer.android.com/reference/android/app/Service.html`.

The most important service classes are the following three:

- `android.app.Service`: This is the one we've been using so far. This is the most basic service class. Unless you use multithreading *inside* the service class or the service is explicitly configured to execute in another process, the service will be running inside the service caller's main thread. If this is the GUI thread and you don't expect the service invocation to run really fast, it is strongly recommended you send service activities to a background thread.
- `android.app.IntentService`: While a service by design does not naturally handle incoming start requests simultaneously to the main thread, an `IntentService` uses a dedicated worker thread to receive multiple start messages. Still, it uses just one thread to work with start requests, so they get executed one after the other. `IntentService` classes take care of correctly stopping services, so you don't need to care about this yourself. You have to provide the service's work to be done for each start request inside an overwritten

`onHandleIntent()` method. Since basically you don't need anything else, the `IntentService` service is easy to implement. Note that starting with Android 8.0 (API level 26), restrictions apply to background processes, so under appropriate circumstances, consider using `JobIntentService` classes instead.

- `android.support.v4.app.JobIntentService`: This uses a `JobScheduler` to enqueue service execution requests. Starting with Android 8.0 (API level 26), consider using this service base class for background services. To implement such a service, you basically have to create a subclass of `JobIntentService` and override the method `onHandleWork(intent: Intent): Unit` to contain the service's workload.

Services Lifecycle

Having described various service characteristics in the preceding sections, the actual lifecycle of a service from a bird's-view perspective is arguably easier than that of an activity. However, be careful of services being able to run in the background. Also, because services are more readily subject to stops forced by the Android OS, they may require special attention in correspondence with service clients.

In your service implementation, you can overwrite any the lifecycle callbacks listed here, for example, to log service invocation information while developing or debugging:

- `onCreate()`
- `onStartCommand()`
- `onBind()`
- `onUnbind()`
- `onRebind()`
- `onDestroy()`

Figure 4-1 shows an overview of the lifecycle of a service

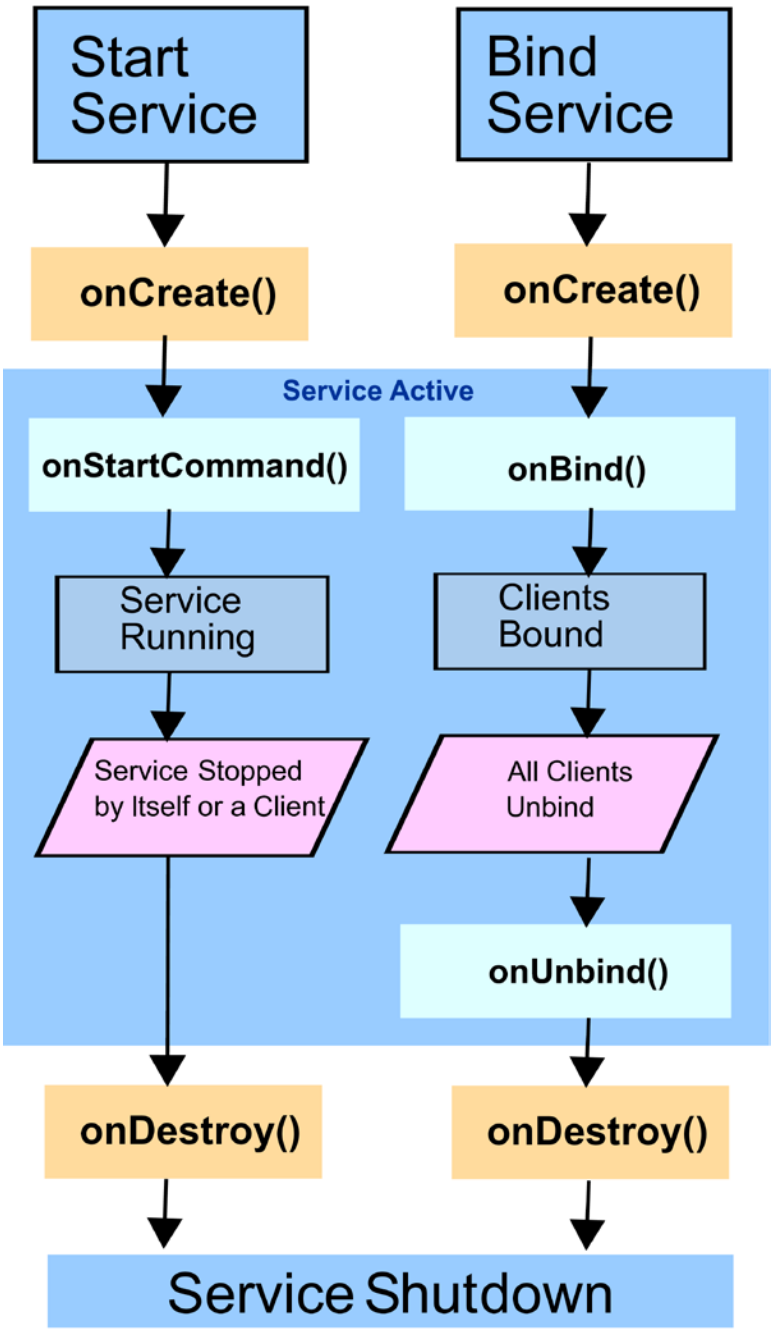

Figure 4-1. *Service lifecycle*

More Service Characteristics

The following are more observations about services:

- Services get declared alongside activities inside `AndroidManifest.xml`. A common question is how they interact with each other. Somebody needs to invoke services to use them, but this can also be done from other services, other activities, or even other apps.
- Do not bind or unbind during an activity's `onResume()` and `onPause()` methods for performance and stability reasons. Do bind and unbind instead inside the `onStart()` and `onStop()` methods, if you need to interact with services only when an activity is visible. If you need service connections also when activities are stopped and in the background, do bind and unbind in the `onCreate()` and `onRestore()` methods.
- In remote connection operations (the service lives in another app), catch and handle `DeadObjectException` exceptions.
- If you overwrite a service's `onStartCommand(intent: Intent, flags: Int, startId: Int)` method, first make sure to also call the method `super.onStartCommand()` unless you have good reasons not to do that. Next, appropriately react on the incoming `flags` parameter, which tells whether this is an automatic follow-up start request because a previous start attempt failed. Eventually this method returns an integer describing the service's state after leaving the `onStartCommand()` method; see the API documentation for details.
- Calling `stopService()` from outside a service or `stopSelf()` from inside a service does not guarantee that the service gets stopped immediately. Expect the service to hang around for a little while until Android really stops it.
- If a service is not designed to react on binding requests and you overwrite the `onBind()` method of the service, it should return `null`.
- While not forbidden explicitly, for a service that is designed for communicating with service clients via binding, consider disallowing the service to be started by `startService()`. In this case, you *must* provide the `Context.BIND_AUTO_CREATE` flag in the `bindService()` method call.

Chapter 5

Broadcasts

Android broadcasts are messages following the publish-subscribe pattern. They are sent across the Android OS, with the internals hidden by the Android OS, so both publishers and subscribers see only a lean asynchronous interface for sending and receiving messages. Broadcasts can be published by the Android OS itself, by standard apps, and by any other app installed on the system. Likewise, any app can be configured or programmed to receive the broadcast messages they are interested in. Like activities, broadcasts can be explicitly or implicitly routed, which is the responsibility of the broadcast sender to decide.

Broadcast receivers are declared either in the `AndroidManifest.xml` file or programmatically. Starting with Android 8.0 (API level 26), the developers of Android have abandoned the usual symmetry between XML and programmatic declaration of broadcast receivers for implicit intents. The reason is that the general idea of imposing restrictions on processes running in background mode, especially related to broadcasts, resulted in a high load on the Android OS, slowing devices down considerably and leading to a bad user experience. For that reason, the declaration of broadcast receivers inside `AndroidManifest.xml` is now limited to a smaller set of use cases.

> **Note** You will want to write modern apps that are runnable in Android 8.0 and newer. For that reason, take this broadcast limit for implicit intents seriously and make your app live within that limitation.

P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_5

Explicit Broadcasts

An *explicit broadcast* is a broadcast published in such a way that there is exactly one receiver addressed by it. This usually makes sense only if both broadcast publishers and subscribers are part of the same app or, less frequently, part of the same app collection if there is a strong functional dependency among them.

There are differences between local and remote broadcasts: local broadcast receivers *must* reside in the same app, they run fast, and receivers cannot be declared inside `AndroidManifest.xml`. Instead, a programmatical registration method must be used for local broadcast receivers. Also, you must use the following to send local broadcast messages:

```
// send local broadcast
LocalBroadcastManager.getInstance(Context).
      sendBroadcast(...)
```

Remote broadcast receivers, on the other hand, *can* reside in the same app, they are slower, and it is possible to use `AndroidManifest.xml` to declare them. To send remote broadcasts, you write the following:

```
// send remote broadcast (this App or other Apps)
sendBroadcast(...)
```

> **Note** Local broadcasts should be favored over remote broadcasts for performance reasons. The apparent disadvantage of not being able to use `AndroidManifest.xml` to declare local receivers does not matter too much, since starting with Android 8.0 (API level 26) the use cases of declaring broadcast receivers inside the manifest files are limited anyway.

Explicit Local Broadcasts

To send a local broadcast message to a local broadcast receiver inside the same app, you write the following:

```
val intent = Intent(this, MyReceiver::class.java)
intent.action = "de.pspaeth.simplebroadcast.DO_STH"
intent.putExtra("myExtra", "myExtraVal")
Log.e("LOG", "Sending broadcast")
LocalBroadcastManager.getInstance(this).
      sendBroadcast(intent)
Log.e("LOG", "Broadcast sent")
```

Here, `MyReceiver` is the receiver class.

```
class MyReceiver : BroadcastReceiver() {
  override
  fun onReceive(context: Context?, intent: Intent?) {
    Toast.makeText(context, "Intent Detected.",
                   Toast.LENGTH_LONG).show()
    Log.e("LOG", "Received broadcast")
```

```
    Thread.sleep(3000)
    // or real work of course...
    Log.e("LOG", "Broadcast done")
  }
}
```

For local broadcasts, the receiver *must* be declared inside the code. To avoid a resource leakage, we create and register the receiver inside `onCreate()` and unregister it inside `onDestroy()`.

```
class MainActivity : AppCompatActivity() {
    private var bcReceiver:BroadcastReceiver? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // ...

        bcReceiver = MyReceiver()
        val ifi:IntentFilter =
              IntentFilter("de.pspaeth.myapp.DO_STH")
        LocalBroadcastManager.getInstance(this).
              registerReceiver(bcReceiver, ifi)
    }

    override fun onDestroy() {
        super.onDestroy()
        // ...

        LocalBroadcastManager.getInstance(this).
            unregisterReceiver(bcReceiver)
    }
}
```

Explicit Remote Broadcasts

We already pointed out that we can send broadcast messages of the *remote* type to other apps or to the same app where the receivers live. The difference is in how the data is sent. For remote messages, the data goes through an IPC channel. Now to send such remote broadcast messages to the same app, you write the following:

```
val intent = Intent(this, MyReceiver::class.java)
intent.action = "de.pspaeth.myapp.DO_STH"
intent.putExtra("myExtra", "myExtraVal")
sendBroadcast(intent)
```

On the receiving side, for remote messages, the receiver *must* be declared inside the manifest file.

```
<application ...>
  ...
  <receiver android:name=".MyReceiver">
    <intent-filter>
      <action android:name=
            "de.pspaeth.myapp.DO_STH">
      </action>
    </intent-filter>
  </receiver>
</application>
```

Looking at the differences between local and remote broadcasts, it is helpful to keep the following in mind:

- **Local explicit broadcasts:**

  The sender uses an explicit receiver class, the receiver must be declared programmatically, and both the sender and receiver use `LocalBroadcastManager` to send messages and to register the receiver.

- **Remote explicit broadcasts:**

  The sender uses explicit receiver class, and the receiver must be declared in `AndroidManifest.xml`.

For the class that is responsible for handling received broadcasts, there is no difference compared to the explicit local broadcasts.

```
class MyReceiver : BroadcastReceiver() {
  override
  fun onReceive(context: Context?, intent: Intent?) {
    // handle incoming broadcasts...
  }
}
```

Explicit Broadcasts Sending to Other Apps

The senders and receivers of explicit broadcasts can live in different apps. For this to work, you can no longer use the intent constructor we used earlier.

```
val intent = Intent(this, MyReceiver::class.java)
intent.action = "de.pspaeth.myapp.DO_STH"
// add other coords...
sendBroadcast(intent)
```

This is because the receiving class, here `MyReceiver`, is not part of the classpath. There is, however, another construct we can use instead.

```
val intent = Intent()
intent.component = ComponentName("de.pspaeth.xyz",
      "de.pspaeth.xyz.MyReceiver")
intent.action = "de.pspaeth.simplebroadcast.DO_STH"
// add other coords...
sendBroadcast(intent)
```

Here, the first argument to `ComponentName` is the package string of the receiving package, and the second argument is the class name.

> **Caution** Unless you are broadcasting to apps that have been built by yourself, this way of sending explicit broadcasts is of limited use only. The developer of the other app may easily decide to change class names, and then your communication to the other app using broadcasts will be broken.

Implicit Broadcasts

Implicit broadcasts are broadcasts with an undefined number of possible receivers. For explicit broadcasts, you learned that we had to build the corresponding intents by using the constructor that points to the recipient component: `val intent = Intent(this, TheReceiverClass::class.java)`. Contrary to that, for implicit broadcast we no longer specify the recipient but instead give hints on which components might be interested in receiving it. Here's an example:

```
val intent = Intent()
intent.action = "de.pspaeth.myapp.DO_STH"
sendBroadcast(intent)
```

Here, we actually express the following: "Send a broadcast message to all receivers that are interested in action `de.pspaeth.myapp.DO_STH`." The Android OS determines which components are eligible to receive such broadcast messages then; this might result in zero, one, or many actual recipients.

There are three decisions you must make before you start programming implicit broadcasts.

- **Do we want to listen to system broadcasts?**

  A large number of predefined broadcast message types exist for Android. Inside the Android SDK that you installed with Android Studio, at `SDK_INST_DIR/platforms/VERSION/data/broadcast_actions.txt`, you can find a list of system broadcast actions. If we want to listen to such messages, we just need to program the appropriate broadcast receivers as described later in the chapter. In the "System Broadcasts" section of the online text companion, you'll find a comprehensive list of the system broadcasts.

- **How do we classify broadcast message types?**

  Broadcast senders and broadcast receivers join by intent filter matches, just like activities do. As discussed in Chapter 3 when describing the intent filters for activities, the classification is threefold for broadcasts: first you have an obligatory action specifier, second a category, and third a data-and-type specifier that you can use to define filters. We describe this matching procedure later in this chapter.

- **Are we heading for local or remote broadcasts?**

  If all the broadcasting happens completely inside your app, you should use local broadcasting for sending and receiving messages. For implicit broadcasts, this will probably not be the case too often, but for large complex apps, this is totally acceptable. If system broadcasts or broadcasts from other apps are involved, you *must* use remote broadcasts. The latter is the default case in most examples, so you will see this pattern quite often.

Intent Filter Matching

Broadcast receivers express their accepting broadcasts by means of declaring *action*, *category*, and *data specifiers*.

Let's first talk about *actions*. These are just strings without any syntax restriction. Looking more thoroughly at them, you see that we first have a more or less stringently defined set of predefined action names. We listed them all in Chapter 3. In addition, you can define your own actions. A convention is to use your package name plus a dot and then an action specifier. You are not forced to follow this convention, but it is strongly recommended to do it that way so your apps do not get messed up with other apps. Without specifying any other filter criteria, a sender specifying that particular action you specified in the filter will reach all matching receivers.

- For an intent filter to match, the *action* specified on the receiver side must match the *action* specified on the sender side. For implicit broadcasts, zero, one, or many receivers might be addressed by one broadcast.
- A receiver may specify more than one filter. If one of the filters contains the specified *action*, this particular filter will match the broadcast.

Table 5-1 shows some examples.

***Table 5-1.** Action Matches*

Receiver	Sender	Match
One filter `action = "com.xyz.ACTION1"`	`action = "com.xyz.ACTION1"`	Yes
One filter `action = "com.xyz.ACTION1"`	`action = "com.xyz.ACTION2"`	No

Two filters
`action = "com.xyz.ACTION1"`
`action = "com.xyz.ACTION2"`
	`action = "com.xyz.ACTION1"`	Yes

Two filters
`action = "com.xyz.ACTION1"`
`action = "com.xyz.ACTION2"`
	`action = "com.xyz.ACTION3"`	No

Besides *actions*, a *category* specifier can be used to restrict an intent filter. We have a couple of predefined categories listed in Chapter 3, but again you can define your own categories. Like for *actions*, for your own *categories* you should follow the naming convention of prepending your app's package name to your category name. Once during the intent matching process a match in the *action* is found, all the categories that were declared by the sender must be present as well in the receiver's intent filter for the match to further prevail.

- Once an *action* inside an intent filter matches a broadcast and the filter also contains a list of categories, only such broadcasts will match the filter for which the categories specified by the sender are all contained in the receiver's category list.

Table 5-2 shows some examples (one filter only; if there are several filters, the matching happens on an "or" basis).

***Table 5-2.** Category Matches*

Receiver Action	Receiver Category	Sender	Match
com.xyz.ACT1	com.xyz.cate1	action = "com.xyz.ACT1"	Yes
com.xyz.ACT1		action = "com.xyz.ACT1"	No
		categ = "com.xyz.cate1"	
com.xyz.ACT1	com.xyz.cate1	action = "com.xyz.ACT1"	Yes
		categ = "com.xyz.cate1"	
com.xyz.ACT1	com.xyz.cate1	action = "com.xyz.ACT1"	No
		categ = "com.xyz.cate1"	
		categ = "com.xyz.cate2"	
com.xyz.ACT1	com.xyz.cate1	action = "com.xyz.ACT1"	Yes
	com.xyz.cate2	categ = "com.xyz.cate1"	
		categ = "com.xyz.cate2"	
com.xyz.ACT1	com.xyz.cate1	action = "com.xyz.ACT1"	Yes
	com.xyz.cate2	categ = "com.xyz.cate1"	
com.xyz.ACT1	any	action = "com.xyz.ACT2"	No
		categ = any	

Third, a *data-and-type* specifier allows for filtering data types. Such a specifier is one of the following:

- **type**: The MIME type, for example "text/html" or "text/plain"
- **data**: A data URI, for example "http://xyz.com/type1"
- **data** and **type**: Both of them

Here, the data element allows for wildcard matching.

- **Presumed *action* and *category* match:** A *type* filter element matches if the sender's specified MIME type is contained in the receiver's list of allowed MIME types.
- **Presumed *action* and *category* match:** A *data* filter element matches if the sender's specified data URI matches any of the receiver's list of allowed data URIs (wildcard matching might apply).
- **Presumed *action* and *category* match:** A *data-and-type* filter element matches if both the MIME type and the data URI match, i.e., are contained within the receiver's specified list.

Table 5-3 shows some examples (one filter only; if there are several filters, the matching happens on an "or" basis).

Table 5-3. *Data Matches*

Receiver Type	Receiver URI .* = any string	Sender	Match
text/html		type = "text/html"	Yes
text/html text/plain		type = "text/html"	Yes
text/html text/plain		type = "image/jpeg"	No
	http://a.b.c/xyz	data = "http://a.b.c/xyz"	Yes
	http://a.b.c/xyz	data = "http://a.b.c/qrs"	No
	http://a.b.c/xyz/.*	data = "http://a.b.c/xyz/3"	Yes
	http://.*/xyz	data = "http://a.b.c/xyz"	Yes
	http://.*/xyz	data = "http://a.b.c/qrs"	No
text/html	http://a.b.c/xyz/.*	type = "text/html" data = "http://a.b.c/xyz/1"	Yes
text/html	http://a.b.c/xyz/.*	type = "image/jpeg" data = "http://a.b.c/xyz/1"	No

Active or On-Hold Listening

Which state must an app be in to be able to receive implicit broadcasts? If we want a broadcast receiver to be just registered in the system and fired up only on demand when a matching broadcast arrives, the listener must be specified in the manifest file of the app. However, for implicit broadcasts, this cannot be freely done. It is only for predefined system broadcasts, as listed in section "System Broadcasts" of the online text companion.

> **Note** This restriction for implicit intent filters specified in the manifest was introduced in Android 8.0 (API level 26). Before that, any implicit filters could be specified inside the manifest file.

If, however, you start your broadcast listeners programmatically from inside an app and this app is running, you can define as many implicit broadcast listeners as you want, and there is no restriction on whether the broadcasts come from the system, your app, or other apps. Likewise, there is no restriction on usable *action* or *category* names.

Since listening for booting-completed events is included in the list for allowed listeners inside the manifest file, you are free to start apps there as activities or services, and inside those apps you can register any implicit listener. But that means you can legally work around the restrictions imposed starting with Android 8.0. Just be aware that such apps may be killed by the Android OS if resource shortages occur, so you have to take appropriate precautions.

Sending Implicit Broadcasts

To prepare to send an implicit broadcast, you specify actions, categories, data, and extra data as follows:

```
val intent = Intent()
intent.action = "de.pspaeth.myapp.DO_STH"
intent.addCategory("de.pspaeth.myapp.CATEG1")
intent.addCategory("de.pspaeth.myapp.CATEG2")
// ... more categories
intent.type = "text/html"
intent.data = Uri.parse("content://myContent")
intent.putExtra("EXTRA_KEY", "extraVal")
intent.flags = ...
```

Only the action is mandatory; all the others are optional. Now to send the broadcast, you'd write the following for a remote broadcast:

```
sendBroadcast(intent)
```

For a local broadcast, you'd write this:

```
LocalBroadcastManager.getInstance(this).
      sendBroadcast(intent)
```

The `this` must be a `Context` or a subclass thereof; it will work exactly like shown here if the code is from inside an activity or a service class.

For remote messages, there is also a variant sending a broadcast to applicable receivers one at a time.

```
...
sendOrderedBroadcast(...)
...
```

This makes the receivers get the message sequentially, and each receiver may cancel, forwarding the message to the next receiver in the line by using `BroadcastReceiver.abortBroadcast()`.

Receiving Implicit Broadcasts

To receive an implicit broadcast, for a limited set of broadcast types (see the section "The System Intent Filters" of the online text companion) you can specify a `BroadcastListener` inside `AndroidManifest.xml` as follows:

```
<application ...>
  ...
  <receiver android:name=".MyReceiver">
    <intent-filter>
      <action android:name=
          "com.xyz.myapp.DO_STH" />
      <category android:name=
          "android.intent.category.DEFAULT"/>
      <category android:name=
          "com.xyz.myapp.MY_CATEG"/>
      <data android:scheme="http"
            android:port="80"
            android:host="com.xyz"
            android:path="items/7"
            android:mimeType="text/html" />
      </intent-filter>
    </receiver>
</application>
```

The `<data>` element shown here is just an example; see Chapter 3 for all the possibilities.

In contrast to that, adding a programmatic listener for implicit broadcasts to your code is unrestricted.

```
class MainActivity : AppCompatActivity() {
  private var bcReceiver:BroadcastReceiver? = null

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // ...
    bcReceiver = MyReceiver()
    val ifi:IntentFilter =
    IntentFilter("de.pspaeth.myapp.DO_STH")
    registerReceiver(bcReceiver, ifi)
  }

  override fun onDestroy() {
    super.onDestroy()
    // ...
    unregisterReceiver(bcReceiver)
  }
}
```

The `MyReceiver` is an implementation of class `android.content.BroadcastReceiver`.

Listening to System Broadcasts

To listen to system broadcasts, see the list in the online text companion's "System Broadcasts" section. You can just use a programmatic registration as shown earlier. For most of them, you cannot use the manifest registration method for background execution limits imposed since Android 8.0 (API level 26). However, for a number of them, you can also use the manifest file to specify listeners.

- `ACTION_LOCKED_BOOT_COMPLETED`, `ACTION_BOOT_COMPLETED`:

  Apps may need those to schedule jobs, alarms, and so on.

- `ACTION_USER_INITIALIZE`, `"android.intent.action.USER_ADDED"`, `"android.intent.action.USER_REMOVED"`:

  These are protected by privileged permissions, so the use cases are limited.

- `"android.intent.action.TIME_SET"`, `ACTION_TIMEZONE_CHANGED`, `ACTION_NEXT_ALARM_CLOCK_CHANGED`:

  These are needed by clock apps.

- `ACTION_LOCALE_CHANGED`:

  The locale changed, and apps might need to update their data when this happens.

- `ACTION_USB_ACCESSORY_ATTACHED`, `ACTION_USB_ACCESSORY_DETACHED`, `ACTION_USB_DEVICE_ATTACHED`, `ACTION_USB_DEVICE_DETACHED`:

  These are USB-related events.

- `ACTION_CONNECTION_STATE_CHANGED`, `ACTION_ACL_ CONNECTED`, `ACTION_ACL_DISCONNECTED`:

  These are Bluetooth events.

- `ACTION_CARRIER_CONFIG_CHANGED`, `TelephonyIntents.ACTION_*_SUBSCRIPTION_CHANGED`, `"TelephonyIntents.SECRET_CODE_ACTION"`:

  OEM telephony apps may need to receive these broadcasts.

- `LOGIN_ACCOUNTS_CHANGED_ACTION`:

  This is needed by some apps to set up scheduled operations for new and changed accounts.

- `ACTION_PACKAGE_DATA_CLEARED`:

  Data is cleared by the OS Settings app; a running app likely is interested in that.

- `ACTION_PACKAGE_FULLY_REMOVED`:

  Related apps might need to be informed if some apps get uninstalled and their data is removed.

- `ACTION_NEW_OUTGOING_CALL`:

  This intercepts outgoing calls.

- `ACTION_DEVICE_OWNER_CHANGED`:

  Some apps might need to receive this so that they know the device's security status has changed.

- `ACTION_EVENT_REMINDER`:

  This is sent by the calendar provider to post an event reminder to the calendar app.

- `ACTION_MEDIA_MOUNTED`, `ACTION_MEDIA_CHECKING`, `ACTION_MEDIA_UNMOUNTED`, `ACTION_MEDIA_EJECT`, `ACTION_MEDIA_UNMOUNTABLE`,`ACTION_MEDIA_REMOVED`, `ACTION_MEDIA_BAD_REMOVAL`:

  Apps might need to know about the user's physical interactions with the device.

- `SMS_RECEIVED_ACTION`, `WAP_PUSH_RECEIVED_ACTION`:

  These are needed by SMS recipient apps.

Adding Security to Broadcasts

Security in broadcasting messages is handled by the *permission* system, which gets handled in more detail in Chapter 7.

In the following sections, we distinguish between explicit and implicit broadcasts.

Securing Explicit Broadcasts

For nonlocal broadcasting (i.e., not using the `LocalBroadcastManager`), permissions can be specified on both sides, the receiver *and* the sender. For the latter, the broadcast-sending methods have overloaded versions, including a permission specifier:

```
...
val intent = Intent(this, MyReceiver::class.java)
...
sendBroadcast(intent, "com.xyz.theapp.PERMISSION1")
...
```

This expresses sending a broadcast to a receiver that is protected by `com.xyz.theapp.PERMISSION1`. Of course, you should write your own package names here and use appropriate permission names.

Instead, sending a broadcast without a permission specification may address receivers with and without permission protection:

```
...
val intent = Intent(this, MyReceiver::class.java)
...
sendBroadcast(intent)
...
```

This means that specifying permissions on the sender side is not supposed to tell the receiver that the sender is protected in any way.

For adding permissions to the receiver side, we first need to declare using it inside `AndroidManifest.xml` on an app level.

```
<manifest ...>
    <uses-permission android:name=
          "com.xyz.theapp.PERMISSION1"/>
    ...
    <application ...
```

Next we explicitly add it to the receiver element inside the same manifest file.

```
<receiver android:name=".MyReceiver"
        android:permission="com.xyz.theapp.PERMISSION1">
    <intent-filter>
        <action android:name=
                    "com.xyz.theapp.DO_STH" />
    </intent-filter>
</receiver>
```

Here, `MyReceiver` is an implementation of `android.content.BroadcastReceiver`.

Third, since this is a custom permission, you have to declare itself in the manifest file.

```
<manifest ...>
    <permission android:name=
          "com.xyz.theapp.PERMISSION1"/>
    ...
```

The `<permission>` allows for a couple of more attributes; see the section “Manifest Top Level Entries” in the online text companion to learn more about the protection level. The details for and implications of it are explained thoroughly in Chapter 7.

For noncustom permissions, you don’t need to use the `<permission>` element.

> **Caution** Specifying a permission on the sender side without having a matching permission on the receiver side silently fails when you try to send a broadcast. There are also no logging entries, so be careful with sender-side permissions.

If you use local broadcasts with the `LocalBroadcastManager`, you cannot specify permissions on the sender or the receiver side.

Securing Implicit Broadcasts

Like nonlocal explicit broadcasts, the permissions in implicit broadcasts can be specified on both the broadcast sender and the receiver side. On the sender side, you would write the following:

```
val intent = Intent()
intent.action = "de.pspaeth.myapp.DO_STH"
// ... more intent coordinates
sendBroadcast(intent, "com.xyz.theapp.PERMISSION1")
```

This expresses sending a broadcast to all matching receivers that are additionally protected by `com.xyz.theapp.PERMISSION1`. Of course, you should write your own package names here and use the appropriate permission names. As for the usual sender-receiver matching procedure for implicit broadcasts, adding a permission kind of serves as an additional matching criterion, so if there are several receiver candidates looking at the intent filters, for actually receiving this broadcast, only those will be picked out that additionally provide this permission flag.

One more thing that needs to be taken care of for implicit broadcasts is specifying the permission usage in `AndroidManifest.xml`. So, for this sender to be able to use the permission, add the following to the manifest file:

```
 <uses-permission android:name="com.xyz.theapp.
PERMISSION1"/>
```

It's the same as for explicit broadcasts. Sending a broadcast without a permission specification may address receivers with and without permission protection.

```
...
sendBroadcast(intent)
...
```

This means specifying permissions on the sender side is not supposed to tell the receiver that the sender is protected in any way.

For a receiver to be able to get hold of such a broadcast, the permission must be added to the code like this:

```
private var bcReceiver: BroadcastReceiver? = null

override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  ...
  bcReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context?,
          intent: Intent?) {
```

```
      // do s.th. when receiving...
    }
  }
  val ifi: IntentFilter =
         IntentFilter("de.pspaeth.myapp.DO_STH")
  registerReceiver(bcReceiver, ifi,
        "com.xyz.theapp.PERMISSION1", null)
  }

  override fun onDestroy() {
    super.onDestroy()
    unregisterReceiver(bcReceiver)
}
```

In addition, you must both define the permission and declare using it in the receiver's manifest file.

```
...
<uses-permission android:name=
      "com.xyz.theapp.PERMISSION1" />
<permission android:name=
      "com.xyz.theapp.PERMISSION1" />
...
```

Again, for noncustom permissions, you don't need to use the `<permission>` element. For more about permissions, see Chapter 7.

> **Note** As an additional means to improve security, in applicable cases you can use `Intent.setPackage()` to restrict possible receivers.

Sending Broadcasts from the Command Line

For devices you can connect to via the *Android Debug Bridge* (ADB), you can use a shell command on your development PC to send a broadcast message (see Chapter 18). Here's an example of sending an action `de.pspaeth.myapp.DO_STH` to the dedicated receiver `MyReceiver` of the package `de.pspaeth.simplebroadcast` (this is an explicit broadcast message):

```
./adb shell am broadcast -a de.pspaeth.myapp.DO_STH \
      de.pspaeth.simplebroadcast MyReceiver
```

To get a complete synopsis of sending broadcasts in this way, you can use the shell as follows:

```
./adb shell am
```

This command will show you all the possibilities to create broadcast messages and do other things using that `am` command.

Random Notes on Broadcasts

Here is some additional information about broadcasts:

- You can register and unregister programmatically managed receivers also in the callback methods `onPause()` and `onResume()`. Obviously, registering and unregistering will then happen more often compared to using the `onCreate()` / `onDestroy()` pair.
- A currently executing `onReceive()` method will upgrade the process priority to "foreground" level, preventing the Android OS from killing the receiving process. It would then happen only under extreme resource shortage conditions.
- If you have long-running processes inside `onReceive()`, you might think of running them on a background thread, finishing `onReceive()` early. However, since the process priority will be reverted to the normal level after finishing `onReceive()`, your background process is more likely to be killed, breaking your app. You can prevent this by using `Context.goAsync()` and then starting an `AsyncTask` (inside at the end you must call `finish()` on the `PendingResult` object you got from `goAsync()` to eventually free resources), or you can use a `JobScheduler`.
- Custom permissions, like we used in the "Securing Implicit Broadcasts" section, get registered when the app gets installed. Because of that, the app defining the custom permissions must be installed prior to the apps using them.
- Be cautious with sending sensitive information through implicit broadcasts. Potentially malicious apps may try to receive them as well. At the least, you can secure the broadcast by specifying permissions on the sender side.
- For clarity and to not mess up with other apps, always use namespaces for broadcast action and permission names.
- Avoid starting activities from broadcasts. This contradicts Android usability principles.

Chapter 6

Content Providers

This chapter will cover content providers.

The Content Provider Framework

The content provider framework allows for the following:

- Using (structured) data provided by other apps
- Providing (structured) data for use by other apps
- Copying data from one app to another
- Providing data to the search framework
- Providing data to special data-related UI widgets
- Doing all that by virtue of a well-defined standardized interface

The data communicated can have a strictly defined structure, such as the rows from a database with defined column names and types, but it can also be files or byte arrays without any semantics associated.

If the requirements of your app concerning data storage do not fit in any of the previous cases, you don't need to implement content provider components. Use the normal data storage options instead.

> **Note** It is not strictly forbidden for an app to provide data to its own components or use its own data provider for accessing content; however, when looking at content providers, you usually think of inter-app data exchange. But if you need it, you always can consider intra-app data exchange patterns as a straightforward special case of inter-app communication.

P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_6

If we want to create content-aware apps, both looking at providing and consuming content, these are the main questions:

- How do apps provide content?
- How do apps access content provided by other apps?
- How do apps handle content provided by other apps?
- How do we secure the provided data?

We will be looking at these topics in the following sections. Figure 6-1 shows an outline.

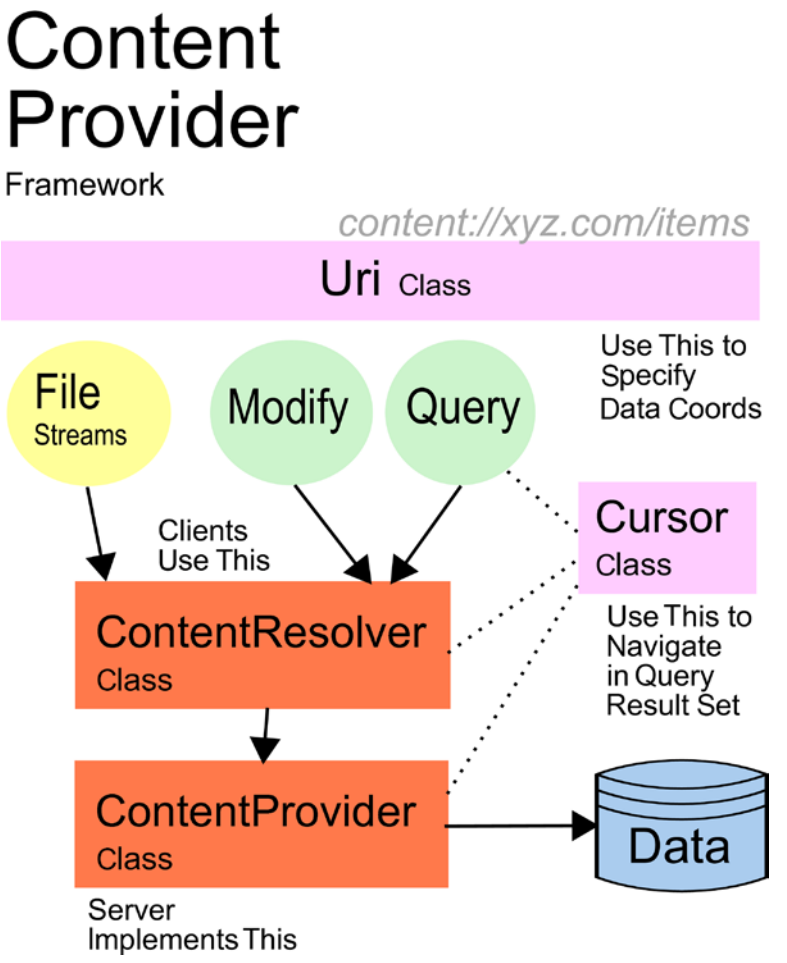

Figure 6-1. *Content provider framework*

Providing Content

Content can be provided by your app as well as by system apps. Think of the pictures taken by the camera or contacts in your contacts list. The content provider framework is a little easier to grasp if we first look at the content-providing side. In later sections, we will also look at the consumers and other topics.

First we need to know where the data lives. However, the content provider framework makes no assumptions on where the data actually comes from. It can reside in files, databases, in-memory storages, or any other place you might think of. This improves the maintenance of your app. For example, in an early stage, data may come from files, but later you may switch to a database or cloud storage, and the possible consumers don't have to care about those changes because they don't have to change how they access your content. The content provider framework thus provides an abstraction layer for your data.

The single interface you need to implement to provide content is the following abstract class:

```
android.content.ContentProvider
```

In the following sections, we will be looking at the implementation of this class from a use case perspective.

Initializing the Provider

You have to implement the following method:

```
ContentProvider.onCreate()
```

This method gets called by the Android OS when the content provider is being instantiated. Here you can initialize the content provider. You should, however, avoid putting time-consuming initialization processes here since instantiation does not necessarily mean the content provider will be actually used.

If you don't have anything interesting to do here, just implement it as an empty method.

To find out more about the environment your content provider is running in when instantiated, you can overwrite its `attachInfo()` method. There you will be told about the context the content provider is running in and also get a `ProviderInfo` object. Just don't forget to also call `super.attachInfo()` from inside.

Querying Data

For querying database-like data sets, there is one method you must implement and two more you can optionally implement.

```
abstract fun query(        // ----- Variant A -----
    uri:Uri,
    projection:Array<String>,
    selection:String,
    selectionArgs:Array<String>,
    sortOrder:String) : Cursor
```

```
                          // ----- Variant B -----
// You don't have to implement this. The default
// implementation calls variant A, but disregards the
// 'cancellationSignal' argument.
fun query(
    uri:Uri,
    projection:Array<String>,
    selection:String,
    selectionArgs:Array<String>,
    String sortOrder:String,
    cancellationSignal:CancellationSignal) : Cursor

                          // ----- Variant C -----
// You don't have to implement this. The default
// implementation converts the bundle argument to
// appropriate arguments for calling variant B.
// The bundle keys used are:
//     ContentResolver.QUERY_ARG_SQL_SELECTION
//     ContentResolver.QUERY_ARG_SQL_SELECTION_ARGS
//     ContentResolver.QUERY_ARG_SQL_SORT_ORDER -or-
//         ContentResolver.QUERY_ARG_SORT_COLUMNS
//         (this being a String array)
fun query(
    uri:Uri,
    projection:Array<String>,
    queryArgs:Bundle,
    cancellationSignal:CancellationSignal) : Cursor
```

These methods are not intended to present file data such as images and sound. Returning links or identifiers to file data is acceptable, though.

In the following list, I describe all the parameters by name and variant:

- `uri`: This is an important parameter specifying the type coordinates of the query in the data space. Content consumers will tell what *kind* of data they are interested in by appropriately specifying this parameter. Since URIs are so important, we describe them in their own section; see "Designing Content URIs" in the "Providing Content" section below. This parameter has the same meaning for variants A, B, and C.
- `projection`: This will tell the implementation which columns the requester is interested in. Looking at the SQL database type of storing data, this lists the column names that should be included in the result. There is, however, no strict requirement for a one-to-one mapping. A requester might ask for a selection parameter `X`, and the values for `X` might be calculated any way you might possibly think of. If `null`, return all fields. This parameter has the same meaning for variants A, B, and C.
- `selection`: This is only for variants A and B. This specifies a selection for the data to be returned. The content provider framework makes no assumptions how this selection parameter must look. It is completely up to the implementation, and content requesters must yield to what the implementation defines. In many cases, you will, however, have something like a SQL selection string like `name = Jean AND age < 45` here. If `null`, return all data sets.

- `selectionArgs`: The selection parameter may contain placeholders like `?`. If so, the values to be inserted for the placeholders are specified in this array. Again, the framework makes no strict assumptions here, but in most cases the ? serves as a placeholder, as in `name = ? AND age < ?`, like in SQL. This may be `null` if there are no selection placeholders.
- `sortOrder`: This is only for variants A and B. This specifies a sort order for the data to be returned. The content provider framework does not prescribe a syntax here, but for SQL-like access, this will usually be something like `name DESC, or ASC`.
- `queryArgs`: This is only for variant C. All three selections, selection arguments, and sort order may or may not be specified using an `android.os.Bundle` object. By convention, for SQL-like queries, the bundle keys are as follows:
  - `ContentResolver.QUERY_ARG_SQL_SELECTION`
  - `ContentResolver.QUERY_ARG_SQL_SELECTION_ARGS`
  - `ContentResolver.QUERY_ARG_SQL_SORT_ORDER`
- `cancellationSignal`: This is only for variants B and C. If this is not `null`, you can use it to cancel the current operation. The Android OS will then appropriately inform the requesters.

All the query methods are supposed to return an `android.database.Cursor` object. This allows you to iterate over the data sets, with the convention that each data set contains an `_id` keyed technical ID. See "A Cursor Class Basing on AbstractCursor" and "Designing Content URIs" in the "Providing Content" section below to learn how to design appropriate cursor objects.

Modifying Content

Content providers do not just allow you to read content, they also allow you to alter content. For this aim, the following methods exist:

```
abstract fun insert(
   Uri uri:Uri,
   values:ContentValues) : Uri

// You don't have to overwrite this, the default
// implementation correctly iterates over the input
// array and calls insert(...) on each element.
fun bulkInsert(
    uri:Uri,
    values:Array<ContentValues>) : Int

abstract fun update(
    uri:Uri,
    values:ContentValues,
    selection:String,
    selectionArgs:Array<String>) : Int
```

```
abstract fun delete(
    uri:Uri,
    selection:String,
    selectionArgs:Array<String>) : Int
```

These are the parameters and their meanings:

- `uri`: This specifies the type coordinates of the data in the data space. Content consumers will tell what *kind* of data they are targeting by appropriately specifying this parameter. Note that for deleting or updating single data sets, it is generally assumed that the URI contains the (technical) ID of the datum at the end of the URI path, for example, `content://com.android.contacts/contact/42`.
- `values`: These are the values to be inserted or updated. You use the various `get*()` methods of this class to access the values.
- `selection`: This specifies a selection for the data to be updated or deleted. The content provider framework makes no assumptions how this selection parameter must look. It is completely up to the implementation, and content requesters must yield to what the implementation defines. In many cases, you will, however, have something like a SQL selection string such as `name = Jean AND age < 45` here. If `null`, all items of the data set will be addressed.
- `selectionArgs`: The selection parameter may contain placeholders like `?`. If so, the values to be inserted for the placeholders are specified in this array. Again, the framework makes no strict assumptions here, but in most cases the `?` serves as a placeholder, as in `name = ? AND age < ?`, like for SQL. It may be `null` if there are no selection placeholders.

The `insert()` method is supposed to return the URI specifying the inserted data. This return value may be `null`, so there is no strict requirement to return something here. If it returns something, this should contain the technical ID. All the `Int`-returning methods are supposed to return the number of affected data sets.

If you don't want the content provider to be able to alter any data, you can just provide empty implementations to all the insert, update, and delete methods, and let them return `0` or `null`.

Finishing the ContentProvider Class

To finish the implementation of your `ContentProvider` class, you must implement one more method in addition to the query, insert, update, and delete methods.

```
abstract getType(uri:Uri) : String
```

This maps any usable URI to the appropriate MIME type. For a possible implementation, you can, for example, use URIs as follows:

```
ContentResolver.CURSOR_DIR_BASE_TYPE + "/vnd.<name>.<type>"
ContentResolver.CURSOR_ITEM_BASE_TYPE + "/vnd.<name>.< type>"
```

The URIs refer to *possibly* many items, or *at most* one item, respectively. For `<name>`, use a globally unique name, either the reverse company domain or the package name, or a prominent part of it. For `<type>`, use an identifier defining the table name or the data domain.

Registering the Content Provider

Once you finish the `ContentProvider` implementation, you must register it inside the `AndroidManifest.xml` file as follows:

```
<provider android:authorities="list"
          android:directBootAware=["true" | "false"]
          android:enabled=["true" | "false"]
          android:exported=["true" | "false"]
          android:grantUriPermissions=["true" | "false"]
          android:icon="drawable resource"
          android:initOrder="integer"
          android:label="string resource"
          android:multiprocess=["true" | "false"]
          android:name="string"
          android:permission="string"
          android:process="string"
          android:readPermission="string"
          android:syncable=["true" | "false"]
          android:writePermission="string" >
    ...
</provider>
```

Table 6-1 describes the attributes.

***Table 6-1.** The <provider> Element*

Attribute (Prepend `android:` to Each)	Description
`authorities`	This is a semicolon-separated list of authorities. In many cases, this will be just one, and you usually use the app (package) name or the full class name of the `ContentProvider` implementation. There is no default, and you must have at least one.
`directBootAware`	This specifies whether the content provider can run before the user unlocks the device. The default is `false`.
`enabled`	This specifies whether the content provider is enabled. The default is `true`.
`exported`	This specifies whether other apps can access contents here. Depending on the architecture of your app, access might be restricted to components from the same app, but usually you want other apps to access the content and thus set this to `true`. Starting with API level 17, the default is `false`. Prior to that, the flag doesn't exist and apps behave like this is set to `true`.

(*continued*)

Table 6-1. (*continued*)

Attribute (Prepend `android:` to Each)	Description
`grantUriPermissions`	This specifies whether permission to other apps accessing content from this provider by URI can be temporarily granted. *Temporarily granted* means a permission denial as defined by `<permission>`, `<readPermission>`, or `writePermission` gets overridden temporarily if the content accessing client was invoked by an intent with `intent.addFlags(Intent.FLAG_GRANT_*_URI_PERMISSION)`. If you set this attribute to `false`, a more fine-grained temporary permission can still be granted by setting one or more `<grant-uri-permission>` subelements. The default is `false`.
`icon`	This specifies the resource ID of an icon to use for the provider. The default is to use the parent component's icon.
`initOrder`	Here you can impose some order for content providers of the same app to be instantiated. Higher numbers get instantiated first. Use this with care, since startup order dependencies might indicate a bad application design.
`label`	This specifies a string resource ID for a label to use. The default is the app's `label`.
`multiprocess`	If set to `true`, an app running in more than one processes possibly will have multiple instances of the content provider running. Otherwise, at most one instance of the content provider will exist. The default is `false`.
`name`	This specifies the fully qualified class name of the `ContentProvider` implementation.
`permission`	This is a convenient way of setting both `readPermission` and `writePermission`. Specifying one of the latter has precedence.
`process`	Specify a process name if you want the content provider to run in another process than the app itself. If it starts with a colon (:), the process will be private to the app; if it starts with a lowercase letter, a global process will be used (permission to do so required). The default is to run in the app's process.
`readPermission`	This is a permission a client must have to read content of the content provider. You can have a client without that permission still access the content by virtue of the `grantUriPermissions` attribute.
`syncable`	This specifies whether data of the content provider should be synchronized with a server.
`readPermission`	This is a permission a client must have to write to content of the content provider. You can have a client without that permission still access the content by virtue of the `grantUriPermissions` attribute.

If you use grantUriPermissions to temporarily give the URI permissions to components from other apps called by an implicit intent, you have to carefully tailor such an intent. First add the flag Intent.FLAG_GRANT_READ_URI_PERMISSION, and then add the URI you want to allow access for inside the intent's data field. Here's an example:

```
intent.action =
      "com.example.app.VIEW" // SET INTENT ACTION
intent.flags =
      Intent.FLAG_ACTIVITY_NEW_TASK
intent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
      // GRANT TEMPORARY READ PERMISSION
intent.data = Uri.parse("content://<AUTHORITY>/<PATH>")
      // USE YOUR OWN!
startActivity(intent)
```

Inside the intent filter of the called component, you then must specify a <data> element, and it must contain an appropriate URI *and* a MIME type. The reason why a MIME type must be specified, although we didn't explicitly state one, is that the Android OS uses the content provider's getType(Uri) method to automatically add the MIME type while the intent gets resolved. Here's an example:

```
<intent-filter>
    <action android:name=
          "de.pspaeth.crcons.VIEW"/>
    <category android:name=
          "android.intent.category.DEFAULT"/>
    <data android:mimeType="*/*"
          android:scheme="content"
          android:host="*"
          android:pathPattern=".*"/>
</intent-filter>
```

The called component is then granted access to this URI in the specified way. After it finishes its work, it is supposed to call revokeUriPermission(String, Uri, Int) to revoke the temporary permission it had been given.

```
revokePermission(getPackageName(), uri,
    Intent.FLAG_GRANT_READ_URI_PERMISSION
    and Intent.FLAG_GRANT_WRITE_URI_PERMISSION)
```

Inside the <provider> element, there are a couple of child elements you can add, listed here:

- meta-data:

```
<meta-data android:name="string"
       android:resource="resource specification"
       android:value="string" />
```

This is where either resource or value must be specified. If you use resource, a resource ID such as @string/someString will assign the resource ID itself to the meta-entry, while using value and @string/someString will assign the *contents* of the resource to the meta-entry.

- grant-uri-permission:

```
<grant-uri-permission android:path="string"
                android:pathPattern="string"
                android:pathPrefix="string" />
```

This grants a specific URI permission (use zero to many instances of that child). Only if the parent's attribute `grantUriPermissions` is set to `false` will this child allow the access to specific URIs. Use exactly one of these attributes: `path` is for the complete URI, `pathPrefix` is for URIs starting with that value, and `pathPattern` allows for wildcards (`X*` is for zero to many repetitions of any character *X*, and `.*` is for zero to many repetitions of any character).

- path-permission:

```
<path-permission android:path="string"
             android:pathPrefix="string"
             android:pathPattern="string"
             android:permission="string"
             android:readPermission="string"
             android:writePermission="string" />
```

To define a subset of data a content provider can serve, you can use this element to specify a path and a required permission. The `path` attribute specifies a complete path, the `pathPrefix` attribute allows matching the initial part of a path, and `pathPattern` is a complete path, but with wildcards (* matches zero to many occurrences of the preceding character, and `.*` matches zero to many occurrences of any character). The `permission` attribute specifies both a read and a write permission, and the attributes `readPermission` and `writePermission` draw a distinction between read and write permission. If one of the latter two is specified, it takes precedence over the `permission` attribute.

Designing Content URIs

URIs describe the domain of the data a content requester is interested in. Thinking of SQL, this would be the table name. URIs can do more, however. The official syntax of a URI is as follows:

```
scheme:[//[user[:password]@]host[:port]]
      [/path][?query][#fragment]
```

You can see that the `user`, `password`, and `port` parts are optional, and in fact you usually wouldn't specify them in an Android environment. They are not forbidden, though, and make sense under certain circumstances. The `host` part, however, is interpreted in the most general way as something that provides something, and this is exactly the way it is interpreted here, with "something" being the data. To make that notion somewhat clearer, the `host` part for Android is commonly referred to as the *authority*. For example, in the Contacts system app, the authority would be `com.android.contacts`. (Don't use strings; use class constant fields instead. See the "Contract" section for more information.) The `scheme` is by convention normally `content`. So, a general contacts URI starts with the following:

```
content://com.android.contacts
```

The path part of the URI specifies the data domain, or table in SQL. The user profile data inside the contacts, for example, gets addressed by the following:

```
content://com.android.contacts/profile
```

In this example, the path has just one element, but in can be more complex like pathpart1/pathpart2/pathpart3.

A URI may also have a query part specifying a selection. Looking at the query methods from the class android.content.ContentProvider, we already have the ability to specify a selection on an API basis, but it is totally acceptable, although not mandatory, to also allow query parameters inside the URI. If you need to put several elements into the query parameter, you can follow the usual convention to use & as a separator, as in name=John&age=37.

The fragment specifies a secondary resource and is not used often for content providers. But you can use it, if it helps you.

Since a URI is such a generic construct and guessing correct URIs for accessing content provided by some apps is almost impossible, a content provider app usually provides a contract class that helps in building correct URIs for the task at hand.

Building a Content Interface Contract

The URIs a client has to use to access data represent the interface to the contents. It is therefore a good idea to have a central place where a client can look to find out which URIs to use. The Android documentation suggests using a content contract class for that purpose. The outline of such a class will look like this:

```
class MyContentContract {
    companion object {
        // The authority and the base URI
        @JvmField
        val AUTHORITY = "com.xyz.whatitis"
        @JvmField
        val CONTENT_URI = Uri.parse("content://" +
                          AUTHORITY)

        // Selection for ID bases query
        @JvmField
        val SELECTION_ID_BASED = BaseColumns._ID +
                           " = ? "

    }
        // For various tables (or item types) it is
        // recommended to use inner classes to specify
        // them. This is just an example
        class Items {
            companion object {
                // The name of the item.
                @JvmField
                val NAME = "item_name"
```

```
                // The content URI for items
                @JvmField
                val CONTENT_URI = Uri.withAppendedPath(
                      MyContentContract.CONTENT_URI, "items")
                // The MIME type of a directory of items
                @JvmField
                val CONTENT_TYPE =
                      ContentResolver.CURSOR_DIR_BASE_TYPE +
                      "/vnd." + MyContentContract.AUTHORITY +
                      ".items"
                // The mime type of a single item.
                @JvmField
                val CONTENT_ITEM_TYPE =
                      ContentResolver.CURSOR_ITEM_BASE_TYPE +
                      "/vnd." + MyContentContract.AUTHORITY +
                      ".items"

              // You could add database column names or
              // projection specifications here, or sort
              // order specifications, and more
              // ...
        }
    }
}
```

Of course, part of your interface design must be using meaningful names for the classes, field names, and field values.

> **Note** The interface described in the contract class does *not* have to correspond to actual database tables. It is completely feasible and beneficial to conceptually decouple the interface from the actual implementation and also provide table joins or other item types here derived in any way you might think of.

Here are a couple of notes about this construct:

- If you can make sure clients will be using only Kotlin as a platform, this can be written in a much shorter way without any boilerplate code.

```
object MyContentContract2 {
  val AUTHORITY = "com.xyz.whatitis"
  val CONTENT_URI = Uri.parse("content://"
                    + AUTHORITY)
  val SELECTION_ID_BASED =
        BaseColumns._ID + " = ? "
  object Items {
      val NAME = "item_name"
      val CONTENT_URI =
            Uri.withAppendedPath(
            MyContentContract.CONTENT_URI, "items")
```

```
        val CONTENT_TYPE =
              ContentResolver.CURSOR_DIR_BASE_TYPE +
              "/vnd." + MyContentContract.AUTHORITY +
              ".items"
        val CONTENT_ITEM_TYPE =
              ContentResolver.CURSOR_ITEM_BASE_TYPE +
              "/vnd." + MyContentContract.AUTHORITY +
              ".items"
    }
}
```

However, if we want Java clients to use the interface as well, we have to use all those companion object and `@JvmObject` declarations and modifiers.

- Using companion objects and `JvmObject` annotations allows for writing `TheClass.THE_FIELD` like for static fields in Java.
- You might consider providing an equivalent Java construct to your clients so they don't have to learn the Kotlin syntax if they use only Java.
- The `Uri.parse()` and `Uri.withAppendedPath()` method calls are just two examples of using the `Uri` class. The class contains a couple more methods that help to manage constructing correct URIs.
- You can also provide helper methods inside the contract class. If you do so, make sure the interface class does not depend on other classes and add a modifier `JvmStatic` to the `fun` function declaration to make it callable from Java.

You would then provide this contract class (or classes, if you want to document the interface using both Kotlin and Java) publicly to any possible clients that are supposed to use your content provider app.

A Cursor Class Based on AbstractCursor and Related Classes

All the `query*()` methods from `ContentProvider` return an `android.database.Cursor` object. From the package you can see that this is a database-centric class, which is actually a small design flaw of Android since the content interface should have been access methodology agnostic.

In addition, the `Cursor` interface is a random access interface for clients wanting to scan through result sets. You can use the base implementation `android.database.AbstractCursor` for your cursor class; it already implements a couple of the interface methods. To do so, write `class MyCursor : AbstractCursor { ... }` or `val myCursor = object : AbstractCursor { ... }` and implement all abstract methods and overwrite some of the other methods for the class to do meaningful things.

- `override fun getCount(): Int`

  This specifies the number of data sets available.

- `override fun getColumnNames(): Array<String>`

  This specifies the ordered array of column names.

- `override fun getInt(column: Int): Int`

  This gets a long value (the column index is zero based).

- `override fun getLong(column: Int): Long`

  This gets a long value (the column index is zero based).

- `override fun getShort(column: Int): Short`

  This gets a short value (the column index is zero based).

- `override fun getFloat(column: Int): Float`

  This gets a float value (the column index is zero based).

- `override fun getDouble(column: Int): Double`

  This gets a double value (the column index is zero based).

- `override fun getString(column: Int): String`

  This gets a string value (the column index is zero based).

- `override fun isNull(column: Int): Boolean`

  This tells whether the value is `null` (the column index is zero based).

- `override fun getType(column: Int): Int`

  You don't have to overwrite this, but if you don't, it will always return `Cursor.FIELD_TYPE_STRING`, assuming that `getString()` will always return something meaningful. For more fine-grained control, let it return one of `FIELD_TYPE_NULL`, `FIELD_TYPE_INTEGER`, `FIELD_TYPE_FLOAT`, `FIELD_TYPE_STRING`, or `FIELD_TYPE_BLOB`. The column index is zero based.

- `override fun getBlob(column: Int): ByteArray`

  Overwrite this, if you want to support blobs. Otherwise, an `UnsupportedOperationException` will be thrown.

- `override fun onMove(oldPosition: Int, newPosition: Int): Boolean`

  Although not marked as `abstract`, you *must* overwrite this. Your implementation must move the cursor to the corresponding position in the result set. Possible values range from `-1` (before the first position; not a valid position) to `count` (after the last position; not a valid position). Let it return `true` if the move was successful. If you don't overwrite it, nothing will happen, and the function returns always `true`.

`AbstractCursor` also provides a method called `fillWindow(position: Int, window: CursorWindow?): Unit` that you can use to fill an `android.database.CursorWindow` object based on the result set from the query. See the online API documentation of `CursorWindow` to proceed with this approach.

Besides `AbstractCursor`, the `Cursor` interface has a couple more (abstract) implementations you can use, as summarized in Table 6-2.

Table 6-2. More Cursor Implementations

Name Inside `android.database`	Description
`AbstractWindowedCursor`	This inherits from `AbstractCursor` and owns a `CursorWindow` object holding the data. Subclasses are responsible for filling the cursor window with data during their `onMove(Int, Int)` operation, allocating a new cursor window if necessary. It's easier to implement compared to `AbstractCursor`, but you have to add a lot of functionality to `onMove()`.
`CrossProcessCursor`	This is a cursor implementation that allows using it from remote processes. It is just an extension of the `android.database.Cursor` interface, containing three more methods: `fillWindow(Int, CursorWindow)`, `getWindow(): CursorWindow`, and `onMove(Int, Int): Boolean`. It does not provide any own implementation; you have to overwrite all the methods defined in `Cursor`.
`CrossProcessCursorWrapper`	This is a cursor implementation that allows using it from remote processes. It implements `CrossProcessCursor` and holds a `Cursor` delegate, which can also be a `CrossProcessCursor`.
`CursorWrapper`	This holds a `Cursor` delegate that all method calls are forwarded to.
`MatrixCursor`	This is a full implementation of `Cursor`, with in-memory storage of data as an `Object` array. You have to use `addRow(...)` to add data. The inner class `MatrixCursor.RowBuilder` can be used to build rows to be used by `MatrixCursor.addRow(Array<Object>)`.
`MergeCursor`	Use this to transparently merge, or concatenate, cursor objects.
`sqlite.SQLiteCursor`	This is an implementation of `Cursor` with the data backed by a SQLite database. Use one of the constructors to connect the cursor with a SQLite database object.

A Cursor Class Based on the Cursor Interface

A more low-level approach of implementing a cursor is not relying on `AbstractCursor` but instead implementing all the interface methods yourself.

You can then use subclassing as in `class MyCursor : Cursor { ... }` or use an anonymous object as in `val myCursor = object : Cursor { ... }`. The section "Cursor Interface" in the online text companion describes all the interface methods.

Dispatching URIs Inside the Provider Code

To simplify dispatching incoming URIs, the class `android.content.UriMatcher` comes in handy. If you have query-related URIs like, for example, this:

```
people                #list all people from a directory
people/37             #inquire a person with ID 37
people/37/phone       #get phone info of person with ID 37
```

and want to use an easy `switch` statement, you can write the following inside your class or object:

```
val PEOPLE_DIR_AUTHORITY = "directory"
val PEOPLE = 1
val PEOPLE_ID = 2
val PEOPLE_PHONES = 3
val uriMatcher = UriMatcher(UriMatcher.NO_MATCH)
init {
    uriMatcher.addURI(PEOPLE_DIR_AUTHORITY,
        "people", PEOPLE)
    uriMatcher.addURI(PEOPLE_DIR_AUTHORITY,
        "people/#", PEOPLE_ID)
    uriMatcher.addURI(PEOPLE_DIR_AUTHORITY,
        "people/#/phone", PEOPLE_PHONES)
}
```

Here, # stands for any number, and * matches any string.

In your `ContentProvider` implementation, you can then use the following construct to dispatch incoming string URLs:

```
when(uriMatcher.match(url)) {
    PEOPLE ->
      // incoming path = people, do s.th. with that...
    PEOPLE_ID ->
      // incoming path = people/#, do s.th. with that...
    PEOPLE_PHONES ->
      // incoming path = people/#/phone, ...
    else ->
      // do something else
}
```

Providing Content Files

Content providers not only can give access to database-like content, they may also expose methods for retrieving file-like data, such as image or sound files. For this aim, the following methods are provided:

- `override fun getStreamTypes(uri:Uri, mimeTypeFilter:String) : Array<String>`

  If your content provider offers files, overwrite this method to allow clients to determine supported MIME types given a URI. The `mimeTypeFilter` should not be `null`, and you can use it to filter the output. It supports wildcards, so if a client wants to retrieve all values, it will write `*/*` here, and your provider code needs to correctly handle this. The output must also contain all those types that may be the result of suitable type conversions performed by the provider. This may return `null` to indicate an empty result set. Examples are `image/png` or `audio/mpeg`.

- `override fun openFile(uri:Uri, mode:String): ParcelFileDescriptor`

  Override this to handle requests to open a file blob. The parameter `mode` must be one of the following (there is no default):

  - `r` for read-only access
  - `w` for write-only access (first erasing if data is already present)
  - `wa`, which is like `w` but possibly appends data
  - `rw` for reading and appending writing
  - `rwt`, which is like `rw` but truncates existing data

  To learn what to do with the returned `ParcelFileDescriptor`, see the text after the list.

- `override fun openFile(uri:Uri, mode:String, signal:CancellationSignal): ParcelFileDescriptor`

  This is the same as `openFile(Uri, String)`, but additionally the client may signal a cancellation while reading the file is in progress. The provider can save the `signal` object and catch the client's cancellation request by periodically calling `throwIfCancelled()` on the `signal` object.

- `override fun openAssetFile(uri:Uri, mode:String): AssetFileDescriptor`

  This is like `openFile(Uri, String)`, but it can be implemented by providers that need to be able to return subsections of files, often assets inside of their APK. For implementing this, you probably want to use the `android.content.res.AssetManager` class. You have it in the `asset` field of a context, so for example in an activity you can directly use `asset` to address the `AssetManager`.

- `override fun openAssetFile(uri:Uri, mode:String, signal:CancellationSignal): AssetFileDescriptor`

  This is the same as `openAssetFile(Uri, String)` but allows for cancellation from the client side. The provider can save the `signal` object and catch the client's cancellation request by periodically calling `throwIfCancelled()` on the `signal` object.

- `override fun : openTypedAssetFile(uri:Uri, mimeTypeFilter:String, opts:Bundle): AssetFileDescriptor`

  Implement this if you want clients to be able to read (not write!) asset data by MIME type. The default implementation compares the `mimeTypeFilter` with whatever it gets from `getType(Uri)`, and if they match, it simply forwards to `openAssetFile(...)`.

- `override fun : openTypedAssetFile(uri:Uri, mimeTypeFilter:String, opts:Bundle, signal:CancellationSignal): AssetFileDescriptor`

  This is the same as `openTypedAssetFile(Uri, String, Bundle)` but allows for cancellation from the client side. The provider can save the `signal` object and catch the client's cancellation request by periodically calling `throwIfCancelled()` on the `signal` object.

- `override fun <T : Any?> openPipeHelper(uri: Uri?, mimeType: String?, opts: Bundle?, args: T, func: PipeDataWriter<T>?): ParcelFileDescriptor`

  This is a helper function for implementing `openTypedAssetFile(Uri, String, Bundle)`. It creates a data pipe and a background thread allowing you to stream generated data back to the client. This function returns a new `ParcelFileDescriptor`. After work is done, the caller must close it.

- `override fun openFileHelper(uri:Uri, mode:String): ParcelFileDescriptor`

  This is a convenience method for subclasses. The default implementation opens a file whose path is given by the result of a `query()` method using the URI provided. For the file path, the `_data` member gets extracted from the query result, and the result set count must be `1`.

Those methods that return a `ParcelFileDescriptor` object can invoke appropriate constructors as follows to build input and output streams for the files:

```
val fd = ... // get the ParcelFileDescriptor
val inpStream =
  ParcelFileDescriptor.AutoCloseInputStream(fd)
val outpStream =
  ParcelFileDescriptor.AutoCloseOutputStream(fd)
```

You must use the `close()` method on the stream once its work is done. The `Auto` means the `ParcelFileDescriptor` gets closed for you automatically when you close the streams.

Similarly, those methods that return an `AssetFileDescriptor` object can invoke appropriate constructors as follows to build input and output streams for the files:

```
val fd = ... // get the AssetFileDescriptor
val inpStream =
  AssetFileDescriptor.AutoCloseInputStream(fd)
val outpStream =
  AssetFileDescriptor.AutoCloseOutputStream(fd)
```

Here again, you must use the `close()` method on the stream once its work is done; only the `AssetFileDescriptor` gets closed for you automatically when you close the streams.

Informing Listeners of Data Changes

A client addressing a content provider via its `ContentResolver` field (e.g., `Activity.contentResolver`) can register to be notified of content changes by calling the following:

```
val uri = ... // a content uri
contentResolver.registerContentObserver(uri, true,
    object : ContentObserver(null) {
      override fun onChange(selfChange: Boolean) {
        // do s.th.
      }
      override fun onChange(selfChange: Boolean,
            uri: Uri?) {
        // do s.th.
      }
    }
)
```

The second argument to `registerContentObserver()` specifies whether sub-URIs (the URI plus any other path elements) will lead to a notification as well. The constructor argument to `ContentObserver` can also be a `Handler` object for receiving `onChange` messages in a different thread.

For this to work, on the content provider side you may need to take care that the event gets correctly emitted. For example, inside any data modification method, you should add the following:

```
context.contentResolver.notifyChange(uri, null)
```

Also, to make change listening bullet-proof, you might want to inform any `Cursor` objects returned by the `query()` methods. For this aim, a `cursor` has a `registerContentObserver()` method that you can use to collect cursor-based content observers. The content provider may then send messages to those content observers as well.

Extending a Content Provider

We have seen that a content provider allows for accessing database-like content and files. If you don't like the way this is done too much or have your own ideas about what a content provider should be able to do, you can implement the `call()` method as follows:

```
override call(method:String, arg:String, extras:Bundle):
 Bundle {
      super.call(method, arg, extras)
      // do your own stuff...
}
```

This way you can design your own content access framework. Of course, you should inform possible clients of how to use the interface, for example, inside the contract class.

> **Caution** No security checks apply to calling this method. You have to implement appropriate security checks yourself, for example by using `checkSelfPermission()` on the context.

Client Access Consistency by URI Canonicalization

Quite often query results contain IDs or list index numbers or other information that depends on some short-term database context. For example, a query may return item IDs like 23, 67, or 56, and if you need to get the details for an item, you query again using another URI containing this ID, for example `content://com.xyz/people/23`. The problem with such URIs is that a client usually wouldn't save them for later retrievals. The ID might have changed meanwhile, and the URI is thus not very reliable.

To overcome this problem, a content provider may implement a *URI canonicalization*. To do so, your content provider class has to implement these two methods:

- `canonicalize(url:Uri): Uri:`

  Let this method return a canonicalized URI, for example, by adding some domain-specific query parameters as follows:

  ```
  content://com.xyz/people/23   ->
  content://com.xyz/people?
        firstName=John&
        lastName=Bird&
        Birthday=20010534&
        SSN=123-99-1624
  ```

- `uncanonicalize(url:Uri): Uri:`

  This does the exact opposite of `canonicalize()`. Let it return `null` if the item gets lost and the uncanonicalization cannot be performed.

Consuming Content

To consume content, content provider clients use an `android.content.ContentResolver` object. Any `Context` object that includes activities, services, and more provides one called `getContentResolver()`, or with Kotlin more concisely addressed by just writing `contentResolver`.

Using the Content Resolver

To access database-like content, you use one of the following `ContentProvider` methods:

- `insert(url: Uri, values: ContentValues): Int`

  This inserts a record.

- `delete(url: Uri, where: String, selectionArgs: Array<String>): Int`

  This deletes records.

- `update(uri: Uri, values: ContentValues, where: String, selectionArgs: Array<String>): Int`

  This updates records.

- `query(uri: Uri, projection: Array<String>, queryArgs: Bundle, cancellationSignal: CancellationSignal): Cursor`

  This queries content according to the parameters given.

- `query(uri: Uri, projection: Array<String>, selection: String, selectionArgs: Array<String>, sortOrder: String, cancellationSignal: CancellationSignal): Cursor`

  This queries content according to the parameters given.

- `query(uri: Uri, projection: Array<String>, selection: String, selectionArgs: Array<String>, sortOrder: String): Cursor`

  This queries content according to the parameters given.

Their signatures and meanings closely relate to the corresponding `ContentProvider` methods, as covered earlier. Also, take a look at the online API reference.

To instead access file content, you can use one of the following methods:

- `openAssetFileDescriptor(uri: Uri, mode: String, cancellationSignal: CancellationSignal): AssetFileDescriptor`

  This opens the inner (asset) file.

- `openAssetFileDescriptor(uri: Uri, mode: String): AssetFileDescriptor`

  This opens the inner (asset) file, with no cancellation signal.

- `openTypedAssetFileDescriptor(uri: Uri, mimeType: String, opts: Bundle, cancellationSignal: CancellationSignal): AssetFileDescriptor`

  This opens the typed inner (asset) file.

- `openTypedAssetFileDescriptor(uri: Uri, mimeType: String, opts: Bundle): AssetFileDescriptor`

  This opens the typed inner (asset) file, with no cancellation signal.

- `openFileDescriptor(uri: Uri, mode: String, cancellationSignal: CancellationSignal): ParcelFileDescriptor`

  This opens the file.

- `openFileDescriptor(uri: Uri, mode: String): ParcelFileDescriptor`

  This opens the file, with no cancellation signal.

- `openInputStream(uri: Uri): InputStream`

  This opens an input stream.

- `openOutputStream(uri: Uri, mode: String): OutputStream`

  This opens an output stream.

- `openOutputStream(uri: Uri): OutputStream`

  This opens an output stream, in `w` mode.

The `open*Descriptor()` methods again closely relate to the corresponding `ContentProvider` methods from the "Providing Content" section. The two others, `openInputStream()` and `openOutputStream()`, are convenience methods to more readily access file (stream) data.

To register content observers for asynchronously being signaled when content changes, as covered earlier, use one of these:

- `registerContentObserver(uri: Uri, notifyForDescendants: Boolean, observer: ContentObserver)`
- `unregisterContentObserver(observer: ContentObserver)`

To use a content provider that exhibits an extension by virtue of an implementation of its `call()` method, you use the corresponding `call()` method of the content resolver

- `call(uri: Uri, method: String, arg: String, extras: Bundle)`

Accessing System Content Providers

The Android OS and its preinstalled apps provide several content provider components. Inside the online API documentation, you can find the content provider contract classes in the "android.provider/Classes" section. The following sections summarize what they are and how they can be accessed.

BlockedNumberContract

This exposes a table containing blocked numbers. Only the system, the default phone app, the default SMS app, and carrier apps can access this table, except for `canCurrentUserBlockNumbers()`, which can be called by any app. To use it, you, for example, write this:

```
val values = ContentValues()
values.put(BlockedNumbers.COLUMN_ORIGINAL_NUMBER,
      "1234567890")
Uri uri = contentResolver.insert(
      BlockedNumbers.CONTENT_URI, values)
```

CalendarContract

This is a rather complex content provider with many tables. As an example, we are accessing the calendars list and adding an event here:

```
val havePermissions =
    ContextCompat.checkSelfPermission(this,
              Manifest.permission.WRITE_CALENDAR)
    == PackageManager.PERMISSION_GRANTED
    && ContextCompat.checkSelfPermission(this,
              Manifest.permission.READ_CALENDAR)
    == PackageManager.PERMISSION_GRANTED
if(!havePermissions) {
    // Acquire permissions...
}else{
    data class CalEntry(val name: String, val id: String)
    val calendars = HashMap<String, CalEntry>()
    val uri = CalendarContract.Calendars.CONTENT_URI
    val cursor = contentResolver.query(
          uri, null, null, null, null)
    cursor.moveToFirst()
    while (!cursor.isAfterLast) {
        val calName = cursor.getString(
            cursor.getColumnIndex(
                CalendarContract.Calendars.NAME))
        val calId = cursor.getString(
            cursor.getColumnIndex(
                CalendarContract.Calendars._ID))
        calendars[calName] = CalEntry(calName, calId)
        cursor.moveToNext()
    }
    Log.e("LOG", calendars.toString())

    val calId = "4" // You should instead fetch an
                    // appropriate entry from the map!
    val year = 2018
    val month = Calendar.AUGUST
    val dayInt = 27
    val hour = 8
    val minute = 30

    val beginTime = Calendar.getInstance()
    beginTime.set(year, month, dayInt, hour, minute)
    val event = ContentValues()
    event.put(CalendarContract.Events.CALENDAR_ID,
          calId)
    event.put(CalendarContract.Events.TITLE,
          "MyEvent")
    event.put(CalendarContract.Events.DESCRIPTION,
          "This is test event")
    event.put(CalendarContract.Events.EVENT_LOCATION,
          "School")
```

```
    event.put(CalendarContract.Events.DTSTART,
          beginTime.getTimeInMillis())
    event.put(CalendarContract.Events.DTEND,
          beginTime.getTimeInMillis())
    event.put(CalendarContract.Events.ALL_DAY,0)
    event.put(CalendarContract.Events.RRULE,
          "FREQ=YEARLY")
    event.put(CalendarContract.Events.EVENT_TIMEZONE,
          "Germany")
    val retUri = contentResolver.insert(
          CalendarContract.Events.CONTENT_URI, event)
    Log.e("LOG", retUri.toString())
}
```

We didn't implement the permission inquiry; permissions are described in detail in Chapter 7.

CallLog

This is a table listing placed and received calls. Here's an example to list the table:

```
val havePermissions =
      ContextCompat.checkSelfPermission(this,
           Manifest.permission.READ_CALL_LOG)
               == PackageManager.PERMISSION_GRANTED
      && ContextCompat.checkSelfPermission(this,
           Manifest.permission.WRITE_CALL_LOG)
               == PackageManager.PERMISSION_GRANTED
if(!havePermissions) {
    // Acquire permissions...
}else {
    val uri = CallLog.Calls.CONTENT_URI
    val cursor = contentResolver.query(
          uri, null, null, null, null)
    cursor.moveToFirst()
    while (!cursor.isAfterLast) {
        Log.e("LOG", "New entry:")
        for(name in cursor.columnNames) {
            val v = cursor.getString(
                  cursor.getColumnIndex(name))
            Log.e("LOG","  > " + name + " = " + v)
        }
        cursor.moveToNext()
    }
}
```

We didn't implement the permission inquiry; permissions are described in detail in Chapter 7. Table 6-3 describes the table columns.

Table 6-3. CallLog Table Columns

Name	Description
date	The date of the call, in milliseconds since the epoch.
transcription	Transcription of the call or voicemail entry.
photo_id	The cached photo ID of an associated photo.
subscription_component_name	The component name of the account used to place or receive the call.
type	The type of the call. One of (constant names in CallLog.Calls): INCOMING_TYPE OUTGOING_TYPE MISSED_TYPE VOICEMAIL_TYPE REJECTED_TYPE BLOCKED_TYPE ANSWERED_EXTERNALLY_TYPE
geocoded_location	A geocoded location for the number associated with this call.
presentation	The number presenting rules set by the network. One of (constant names in CallLog.Calls): PRESENTATION_ALLOWED PRESENTATION_RESTRICTED PRESENTATION_UNKNOWN PRESENTATION_PAYPHONE
duration	The duration of the call in seconds.
subscription_id	The identifier for the account used to place or receive the call.
is_read	Whether this item has been read or otherwise consumed by the user (0=false, 1=true).
number	The phone number as the user entered it.
features	Bitmask describing features of the call, built of (constant names in CallLog.Calls):FEATURES_HD_CALL: Call was HD. FEATURES_PULLED_EXTERNALLY: Call was pulled externally. FEATURES_VIDEO: Call had video. FEATURES_WIFI: Call was WIFI call.
voicemail_uri	URI of the voicemail entry, if applicable.
normalized_number	The cached normalized (E164) version of the phone number, if it exists.
via_number	For an incoming call, the secondary line number the call was received via. When a SIM card has multiple phone numbers associated with it, this value indicates which of the numbers associated with the SIM was called.

(*continued*)

Table 6-3. (*continued*)

Name	Description
matched_number	The cached phone number of the contact that matches this entry, if it exists.
last_modified	The date the row is last inserted, updated, or marked as deleted. In milliseconds since the epoch. Read only.
new	Whether the call has been acknowledged (0=false, 1=true).
numberlabel	The cached number label for a custom number type, associated with the phone number, if it exists.
lookup_uri	The cached URI to look up the contact associated with the phone number, if it exists.
photo_uri	The cached photo URI of the picture associated with the phone number, if it exists.
data_usage	The data usage of the call in bytes.
phone_account_address	Undocumented.
formatted_number	The cached phone number, formatted with rules based on the country the user was in when the call was made or received.
add_for_all_users	Undocumented.
numbertype	The cached number type associated with the phone number, if applicable. One of (constant names in CallLog.Calls): INCOMING_TYPE OUTGOING_TYPE MISSED_TYPE VOICEMAIL_TYPE REJECTED_TYPE BLOCKED_TYPE ANSWERED_EXTERNALLY_TYPE
countryiso	The ISO 3166-1 two-letter country code of the country where the user received or made the call.
name	The cached name associated with the phone number, if it exists.
post_dial_digits	The post-dial portion of a dialed number.
transcription_state_id	Undocumented.
	The (technical) ID of the table entry.

ContactsContract

This is a complex contract describing the phone contacts. Contact information is stored in a three-tier data model.

- ContactsContract.Data:

  Any kind of personal data.

- `ContactsContract.RawContacts`:

  A set of data describing a person.

- `ContactsContract.Contacts`:

  An aggregated view on a person, possibly related to several rows inside the `RawContacts` table. Because of its aggregating nature, it is writable only in parts.

There are more contract-related tables described as inner classes of `ContactsContract`. Instead of explaining all the possible use cases for the contact's content provider, to get you started, we just present code to list the contents of the three main tables listed previously, show what a single new contact writes there, and otherwise refer to the online documentation of the `ContactsContract` class. To list the contents of the three tables, use the following:

```
fun showTable(tbl:Uri) {
   Log.e("LOG", "###################################")
   Log.e("LOG", tbl.toString())
   val cursor = contentResolver.query(
         tbl, null, null, null, null)
   cursor.moveToFirst()
   while (!cursor.isAfterLast) {
      Log.e("LOG", "New entry:")
      for(name in cursor.columnNames) {
         val v = cursor.getString(
               cursor.getColumnIndex(name))
         Log.e("LOG","  > " + name + " = " + v)
       }
       cursor.moveToNext()
   }
}
...
showTable(ContactsContract.Contacts.CONTENT_URI)
showTable(ContactsContract.RawContacts.CONTENT_URI)
showTable(ContactsContract.Data.CONTENT_URI)
```

If you create a new contact using Android's pre-installed Contacts app, inside the `Contacts` view table you will find the following new entry (here only the important columns):

```
_id = 1
display_name_alt = Mayer, Hugo
sort_key_alt = Mayer, Hugo
has_phone_number = 1
contact_last_updated_timestamp = 1518451432615
display_name = Hugo Mayer
sort_key = Hugo Mayer
times_contacted = 0
name_raw_contact_id = 1
```

As an associated entry inside the table RawContacts, you will find among others the following:

```
_id = 1
account_type = com.google
contact_id = 1
display_name_alt = Mayer, Hugo
sort_key_alt = Mayer, Hugo
account_name = pmspaeth1111@gmail.com
display_name = Hugo Mayer
sort_key = Hugo Mayer
times_contacted = 0
account_type_and_data_set = com.google
```

Obviously, you find many of these entries also inside the Contacts view listed earlier. Associated are zero to many entries inside the Data table (with only the most important shown).

```
Entry:
  _id = 3
  mimetype = vnd.android.cursor.item/phone_v2
  raw_contact_id = 1
  contact_id = 1
  data1 = (012) 345-6789

Entry:
  _id = 4
  mimetype = vnd.android.cursor.item/phone_v2
  raw_contact_id = 1
  contact_id = 1
  data1 = (098) 765-4321

Entry:
  _id = 5
  mimetype = vnd.android.cursor.item/email_v2
  raw_contact_id = 1
  contact_id = 1
  data1 = null

Entry:
  _id = 6
  mimetype = vnd.android.cursor.item/name
  raw_contact_id = 1
  contact_id = 1
  data3 = Mayer
  data2 = Hugo
  data1 = Hugo Mayer

Entry:
  _id = 7
  mimetype = vnd.android.cursor.item/nickname
  raw_contact_id = 1
  contact_id = 1
  data1 = null
```

```
Entry:
  _id = 8
  mimetype = vnd.android.cursor.item/note
  raw_contact_id = 1
  contact_id = 1
  data1 = null
```

You can see that the rows inside the `Data` table correspond to edit fields inside the GUI. You see two phone numbers, a first and second name, no nickname, and no e-mail address.

DocumentsContract

This is not a contents contract in the same sense as the other contracts we see here. It corresponds to `android.provider.DocumentsProvider`, which is a subclass of `android.content.ContentProvider`. We will be dealing with document providers later in the chapter.

FontsContract

This is a contract that deals with downloadable fonts and does not correspond to content providers.

MediaStore

The media store handles metadata for all media-related files on both internal and external storage devices. This includes audio files, images, and videos. In addition, it handles files in a usage-agnostic manner. That means media and nonmedia files relate to media files. The root class `android.provider.MediaStore` itself does not contain content provider–specific assets, but the following inner classes do:

- `MediaStore.Audio`

  Audio files. Contains more inner classes for music albums, artists, the audio files themselves, genres, and play lists.

- `MediaStore.Images`

  Images.

- `MediaStore.Videos`

  Videos.

- `MediaStore.Files`

  Files in general.

You can investigate any of the media store tables by scanning through the online API documentation. For your own experiments, you can start with the tables as a whole by watching out for constants `EXTERNAL_CONTENT_URI` and `INTERNAL_CONTENT_URI`, or methods `getContentUri()`, and then sending them through the same code we already used earlier.

```
showTable(MediaStore.Audio.Media.getContentUri(
      "internal"))    // <- other option: "external"

fun showTable(tbl:Uri) {
  Log.e("LOG", "########################################")
  Log.e("LOG", tbl.toString())
  val cursor = contentResolver.query(
        tbl, null, null, null, null)
  cursor.moveToFirst()
  while (!cursor.isAfterLast) {
    Log.e("LOG", "New entry:")
    for(name in cursor.columnNames) {
      val v = cursor.getString(
           cursor.getColumnIndex(name))
      Log.e("LOG","  > " + name + " = " + v)
    }
    cursor.moveToNext()
  }
}
```

Settings

This is a content provider that deals with various global and system-level settings. The following are the main URIs as constants from the contract class:

- `Settings.Global.CONTENT_URI`:

  Global settings. All entries are triples of the following:

  - `_id`
  - `android.provider.Settings.NameValueTable.NAME`
  - `android.provider.Settings.NameValueTable.VALUE`

- `Settings.System.CONTENT_URI`:

  Global system-level settings. All entries are triples of the following:

  - `_id`
  - `android.provider.Settings.NameValueTable.NAME`
  - `android.provider.Settings.NameValueTable.VALUE`

- `Settings.Secure.CONTENT_URI`:

This is a secured system setting. Apps are not allowed to alter it. All entries are triples of the following:

  - `_id`
  - `android.provider.Settings.NameValueTable.NAME`
  - `android.provider.Settings.NameValueTable.VALUE`

To investigate these tables, take a look at the online API documentation of `android.provider.Settings`. It describes all possible settings. To list the complete settings, you can use the same function as earlier for the `ContactsContract` contract class.

```
showTable(Settings.Global.CONTENT_URI)
showTable(Settings.System.CONTENT_URI)
showTable(Settings.Secure.CONTENT_URI)
...
fun showTable(tbl:Uri) {
  Log.e("LOG", "################################")
  Log.e("LOG", tbl.toString())
  val cursor = contentResolver.query(
        tbl, null, null, null, null)
  cursor.moveToFirst()
  while (!cursor.isAfterLast) {
    Log.e("LOG", "New entry:")
    for(name in cursor.columnNames) {
      val v = cursor.getString(
            cursor.getColumnIndex(name))
      Log.e("LOG","  > " + name + " = " + v)
    }
    cursor.moveToNext()
  }
}
```

Your app don't need special permission to read the settings. However, writing is possible only for the `Global` and `System` tables, and you also need a special construct to acquire permission.

```
if(!Settings.System.canWrite(this)) {
    val intent = Intent(
          Settings.ACTION_MANAGE_WRITE_SETTINGS)
    intent.data = Uri.parse(
          "package:" + getPackageName())
    startActivity(intent)
}
```

Usually you acquire permissions by calling the following:

```
ActivityCompat.requestPermissions(this,
      arrayOf(Manifest.permission.WRITE\_SETTINGS), 42)
```

However, when setting permissions, this request gets denied immediately by current Android versions. So, you cannot use it and need to call the intent as shown earlier instead.

To access a certain entry, you can again use constants and methods from the contract class.

```
val uri = Settings.System.getUriFor(
      Settings.System.HAPTIC_FEEDBACK_ENABLED)
Log.e("LOG", uri.toString())
val feedbackEnabled = Settings.System.getInt(
          contentResolver,
          Settings.System.HAPTIC_FEEDBACK_ENABLED)
Log.e("LOG", Integer.toString(feedbackEnabled))
```

```
Settings.System.putInt(contentResolver,
      Settings.System.HAPTIC_FEEDBACK_ENABLED, 0)
```

> **Caution** While it is possible to acquire an individual URI for a certain setting, you should not use the `ContentResolver.update()`, `ContentResolver.insert()`, and `ContentResolver.delete()` methods to alter values. Instead, use the methods provided by the contract class.

SyncStateContract

This contract is used by the browser app, the contacts app, and the calendar app to help synchronize user data with external servers.

UserDictionary

This refers to a content provider that allows you to administer and use predictive input based on a word dictionary. As of API level 23, the user dictionary can be used only from input method editors or the spelling checking framework. For modern apps you should not try to use it from another place. This contract thus plays only an informational role.

VoicemailContract

This contract allows for accessing information referring to voicemail providers. It primarily consists of two tables described by inner classes.

- `VoicemailContract.Status`

A voicemail source app uses this contract to tell the system about its state.

- `VoicemailContract.Voicemails`

  This contains the actual voicemails.

You can list the contents of these tables. For example, for the `Voicemails` table, write the following:

```
val uri = VoicemailContract.Voicemails.CONTENT_URI.
      buildUpon().
      appendQueryParameter(
         VoicemailContract.PARAM_KEY_SOURCE_PACKAGE,
         packageName)
      .build()
showTable(uri)

fun showTable(tbl:Uri) {
  Log.e("LOG", "####################################")
  Log.e("LOG", tbl.toString())
  val cursor = contentResolver.query(
        tbl, null, null, null, null)
```

```
  cursor.moveToFirst()
  while (!cursor.isAfterLast) {
    Log.e("LOG", "New entry:")
    for(name in cursor.columnNames) {
      val v = cursor.getString(
            cursor.getColumnIndex(name))
      Log.e("LOG","  > " + name + " = " + v)
    }
    cursor.moveToNext()
  }
}
```

Adding the `VoicemailContract.PARAM_KEY_SOURCE_PACKAGE` URI parameter is important; otherwise, you'll get a security exception.

Batch-Accessing Content Data

The `android.content.ContentProvider` class allows your implementation to use the following:

```
applyBatch(
      operations: ArrayList<ContentProviderOperation>):
      Array<ContentProviderResult>
```

The default implementation iterates through the list and performs each operation in turn, but you can also override the method to use your own logic. The `ContentProviderOperation` objects provided in the parameter describes the operation to perform. It can be one of update, delete, and insert.

For your convenience, that class provides a builder, which you can use for example as follows:

```
val oper:ContentProviderOperation =
        ContentProviderOperation.newInsert(uri)
             .withValue("key1", "val1")
             .withValue("key2", 42)
       .build()
```

Securing Content

From the moment you declare a content provider inside `AndroidManifest.xml` and export it by setting its `exported` attribute to `true`, other apps are allowed to access the complete contents exposed by the provider.

This might not be what you want for sensitive information. As a remedy, to impose restrictions on the content or part of the content, you add permission-related attributes to the `<provider>` element or its subelements.

You basically have the following options:

1. **Securing all content by one criterion**

   To do so, use the `permission` attribute of `<provider>` as follows:

```
<provider ...
        android:permission="PERMISSION-NAME"
        ... >
    ...
</provider>
```

Here, `PERMISSION-NAME` is a system permission or a permission you defined in the `<permission>` element of the app. If you do it that way, the complete content of the provider is accessible only to such clients that successfully acquired exactly this permission. More precisely, any read or write access requires clients to have this permission. If you need to distinguish between *read* permission and *write* permission, you can instead use the `readPermission` and `writePermission` attributes. If you use a mixture, the more specific attributes win.

- `permission = A → writePermission = A, readPermission = A`
- `permission = A, readPermission = B → writePermission = A, readPermission = B`
- `permission = A, writePermission = B → writePermission = B, readPermission = A`
- `permission = A, writePermission = B, readPermission = C → writePermission = B, readPermission = C`

2. **Securing specific URI paths**

   By using the `<path-permission>` subelement of `<provider>`, you can impose restrictions on specific URI paths.

```
<path-permission android:path="string"
          android:pathPrefix="string"
          android:pathPattern="string"
          android:permission="string"
          android:readPermission="string"
          android:writePermission="string" />
```

   In the `*permission` attributes, you specify the permission name and permission scope, just as described earlier for securing all content by one criterion. For the path specification, you use exactly one of the three possible attributes: `path` is for an exact path match, `path- Prefix` is for matching the start of a path, and `pathPattern` allows for wildcards (`X*` is for zero to many occurrences of any character `X`, and `.*` is for zero to many occurrences of any character). Since you can use several `<path-permission>` elements, you can build a fine-grained permission structure in your content provider.

3. **Permission exemptions**

By using the `grantUriPermission` attribute of the `<provider>` element, you can temporarily grant permissions to components called by intent from the app that owns the content provider. If you set `grantUriPermission` to `true` and the intent for calling the other component gets constructed using the help of this:

```
intent.addFlags(
    Intent.FLAG_GRANT_READ_URI_PERMISSION)
/*or*/
intent.addFlags(
    Intent.FLAG_GRANT_WRITE_URI_PERMISSION)
/*or*/
intent.addFlags(
Intent.FLAG_GRANT_WRITE_URI_PERMISSION and
Intent.FLAG_GRANT_READ_URI_PERMISSION)
```

then the called component will have full access to all content of the provider. You can instead set `grantUriPermission` to `false` and add subelements.

```
<grant-uri-permission android:path="string"
                android:pathPattern="string"
                android:pathPrefix="string" />
```

You then control the exemptions in a more fine-grained way. For both to make sense, you obviously must have restrictions set by *`permission` attributes in effect; otherwise, there is nothing you can have exemptions for. The rules for the `<grant-uri-permission>` element's attributes are as explained earlier: `path` is for an exact path match, `pathPrefix` is for matching the start of a path, and `pathPattern` allows for wildcards (X* is for zero to many occurrences of any character X, and `.*` is for zero to many occurrences of any character).

Providing Content for the Search Framework

The Android search framework provides a feature to users to search any data that is available to them by whatever means and using whatever data source. We will be talking about the search framework in Chapter 8; for now it is important to know that content providers play a role for the following:

- Recent query suggestions
- Custom suggestions

For both of them you provide special content provider subclasses and add them to `AndroidManifest.xml` as any other content provider.

Documents Provider

The documents provider is part of the Storage Access Framework (SAF). It allows for a document-centric view of data access, and it also exhibits a hierarchical super-structure of document directories.

> **Note** The SAF was included in API level 19. As of February 2018, this is the version used for more than 90 percent of active Android devices. You cannot use SAF for devices prior to that, but if you really need to cover the remaining 10 percent, you still can provide documents as normal content mediated by content providers and factor out code that can be used by both the SAF and the legacy provider.

The main idea of a documents provider is that your app provides access to documents, wherever the corresponding data are stored, and otherwise doesn't care about how the documents and the documents structure get presented to the user or other apps. The documents provider data model consists of one to many trees starting at root nodes, with subnodes being either documents or directories spanning subtrees, again with other directories and documents. It thus resembles the structure of data in a file system.

To start with a documents provider, you create a class implementing `android.provider.DocumentsProvider`, which itself is a specialized subclass of `android.content.ContentProvider`. At a bare minimum, you have to implement these methods:

- `override fun onCreate(): Boolean`:

Use this to initialize the documents provider. Since this runs on the app's main thread, you must not perform lengthy operations here. But you can prepare the data access to the provider. This returns `true` if the provider was successfully loaded and `false` otherwise.

- `override fun queryRoots(projection: Array<out String>?): Cursor`:

  This is supposed to query the roots of the provider's data structure. In many cases, the data will fit into one tree, and you thus need to provide just one root, but you can have as many roots as makes sense for your requirements. The `projection` argument may present a list of columns to be included in the result set. The names are the same as the `COLUMN_*` constants inside `DocumentsContract.Root`. It may be `null`, which means return all columns. The method must return cursors with at a maximum the following fields (shown are the constant names from `DocumentsContract.Root`):

  - `COLUMN_AVAILABLE_BYTES` (long): Available bytes under the root. Optional, and may be `null` to indicate `unknown`.
  - `COLUMN_CAPACITY_BYTES` (long): The capacity of the tree at that root, in bytes. Think of a file system capacity. Optional, and may be `null` to indicate `unknown`.
  - `COLUMN_DOCUMENT_ID`: The ID (string) of the directory corresponding to that root. Required.
  - `COLUMN_FLAGS`: Flags that apply to a root (int). A combination of (constants in `DocumentsContract.Root`):
    - `FLAG_LOCAL_ONLY` (local to the device, no network access),
    - `FLAG_SUPPORTS_CREATE` (at least one document under the root supports creating content)

- FLAG_SUPPORTS_RECENTS (root can be queried to show recently changed documents)
- FLAG_SUPPORTS_SEARCH (the tree allows for searching documents)

- COLUMN_ICON (int): Icon resource ID for a root. Required.
- COLUMN_MIME_TYPES (string): Supported MIME types. If more than one, use a newline \n as a separator. Optional, and may be null to indicate support for all MIME types.
- COLUMN_ROOT_ID (string): A unique ID of the root. Required.
- COLUMN_SUMMARY (string): Summary for this root; might be shown to a user. Optional, and may be null to indicate "unknown."
- COLUMN_TITLE (string): Title for the root, might be shown to a user. Required.

If this set of roots changes, you must call ContentResolver.notifyChange with DocumentsContract.buildRootsUri to notify the system.

- override fun queryChildDocuments(parentDocumentId: String?, projection: Array<out String>?, sortOrder: String?): Cursor:

Return the immediate children documents and subdirectories contained in the requested directory. Apps targeting at API level 26 or higher should instead implement fun queryChildDocuments(parentDocumentId: String?, projection: Array<out String>?, queryArgs: Bundle?): Cursor and in this method use the following:

```
override fun queryChildDocuments(
      parentDocumentId: String?,
      projection: Array<out String>?,
      sortOrder: String?): Cursor {
    val bndl = Bundle()
        bndl.putString(
            ContentResolver.QUERY_ARG_SQL_SORT_ORDER,
            sortOrder)
        return queryChildDocuments(
            parentDocumentId, projection, bndl)
    }
```

- override fun queryChildDocuments(parentDocumentId: String?, projection: Array<out String>?, queryArgs: Bundle?): Cursor:

Return the immediate children documents and subdirectories contained in the requested directory. The bundle argument contains query parameters as keys.

```
ContentResolver.QUERY_ARG_SQL_SELECTION
ContentResolver.QUERY_ARG_SQL_SELECTION_ARGS
ContentResolver.QUERY_ARG_SQL_SORT_ORDER -or-
     ContentResolver.QUERY_ARG_SORT_COLUMNS
     (this being a String array)
```

The parentDocumentId is the ID of the directory we want to have listed, and inside projection you can specify the columns that should be returned. Use a list of constants COLUMN_* from DocumentsContract.Document. Or write null to return all columns. The resulting Cursor at a maximum returns the following fields (keys are constants from DocumentsContract.Document):

- COLUMN_DISPLAY_NAME (string): The display name of a document, used as the primary title displayed to a user. Required.
- COLUMN_DOCUMENT_ID (string): The unique ID of a document. Required.
- COLUMN_FLAGS: Flags for the document. A combination of (constant names from DocumentsContract.Document):
  - FLAG_SUPPORTS_WRITE (writing supported)
  - FLAG_SUPPORTS_DELETE (deleting supported)
  - FLAG_SUPPORTS_THUMBNAIL (representation as thumbnail supported)
  - FLAG_DIR_PREFERS_GRID (for directories, if they should be shown as a grid)
  - FLAG_DIR_PREFERS_LAST_MODIFIED (for directories, sorting by "last modified" preferred)
  - FLAG_VIRTUAL_DOCUMENT (a virtual document without MIME type)
  - FLAG_SUPPORTS_COPY (copying supported)
  - FLAG_SUPPORTS_MOVE (moving, inside the tree, supported)
  - FLAG_SUPPORTS_REMOVE (removing from the hierarchical structure, not deleting, supported)
- COLUMN_ICON (int) : A specific icon resource ID for a document. May be null to use the system default.
- COLUMN_LAST_MODIFIED (long): The timestamp when a document was last modified, in milliseconds since January 1, 1970 00:00:00.0 UTC. Required, but may be null if undefined.
- COLUMN_MIME_TYPE (string): The MIME type of a document. Required.
- COLUMN_SIZE (long): Size of a document, in bytes, or null if unknown. Required.
- COLUMN_SUMMARY (string): The summary of a document; may be shown to a user. Optional and may be null.

For network-related operations, you might return data partly and set DocumentsContract.EXTRA_LOADING on the Cursor to indicate you are still fetching additional data. Then, when the network data is available, you can send a change notification to trigger a requery and return the complete contents. To support change notifications, you must fire Cursor.setNotificationUri() with a relevant URI, maybe from DocumentsContract.buildChildDocumentsUri(). Then you can call ContentResolver.notifyChange() with that URI to send change notifications.

- `fun openDocument(documentId: String?, mode: String?, signal: CancellationSignal?): ParcelFileDescriptor:`

Open and return the requested document. This should return a reliable `ParcelFileDescriptor` to detect when the remote caller has finished reading or writing the document. If you block while downloading content, you should periodically check `CancellationSignal.isCanceled()` to abort abandoned open requests. The parameters are `documentId` for the document to return. The `mode` specifies the "open" mode, such as `r`, `w`, or `rw`. Mode `r` should always be supported. The provider should throw `UnsupportedOperationException` if the passing mode is not supported. You may return a pipe or socket pair if the mode is exclusively `r` or `w`, but complex modes like `rw` imply a normal file on disk that supports seeking. The `signal` may be used from the caller if the request should be canceled. May be `null`.

- `override fun queryDocument(documentId: String?, projection: Array<out String>?): Cursor:`

Return metadata for a single requested document. The parameters are `documentId` for the ID of the document to return and `projection` for a list of columns to put into the cursor. Use the constants from `DocumentsContract.Document`. For a list, see the description of the method `queryChildDocuments()`. If you use `null` here, all columns are to be returned.

Inside the file `AndroidManifest.xml`, you register the documents provider almost like any other provider.

```
<provider
    android:name="com.example.YourDocumentProvider"
    android:authorities="com.example.documents"
    android:exported="true"
    android:grantUriPermissions="true"
    android:permission=
          "android.permission.MANAGE_DOCUMENTS">
  <intent-filter>
    <action android:name=
          "android.content.action.DOCUMENTS_PROVIDER"/>
  </intent-filter>
</provider>
```

In the previous queries, we have seen that the `Cursor` object returns flags to indicate that recent documents and searching inside the tree should be supported. For this to work, you must implement one or two more methods in your `DocumentsProvider` implementation.

- `override fun queryRecentDocuments(rootId: String, projection: Array<String>): Cursor:`

  This is supposed to return recently modified documents under the requested root. The returned documents should be sorted by `COLUMN_LAST_MODIFIED` in descending order, and at most 64 entries should be shown. Recent documents do not support change notifications.

- `querySearchDocuments(rootId: String, query: String, projection: Array<String>): Cursor`:

  This is supposed to return documents that match the given query under the requested root. The returned documents should be sorted by relevance in descending order. For slow queries, you can return data in part and set `EXTRA_LOADING` on the cursor to indicate that you are fetching additional data. Then, when the data is available, you can send a change notification to trigger a requery and return the complete contents. To support change notifications, you must use `setNotificationUri(ContentResolver, Uri)` with a relevant `Uri`, maybe from `buildSearchDocumentsUri(String, String, String)`. Then you can call the method `notifyChange(Uri, android.database.ContentObserver, boolean)` with that `Uri` to send change notifications.

Once your documents provider is configured and running, a client component can then use an `ACTION_OPEN_DOCUMENT` or `ACTION_CREATE_DOCUMENT` intent to open or create a document. The Android system picker will be taking care of presenting the appropriate documents to the user; you don't have to provide an own GUI for your documents provider.

Here's an example of such a client access:

```
// An integer you can use to identify that call when the
// called Intent returns
val READ_REQUEST_CODE = 42

// ACTION_OPEN_DOCUMENT used in this example is the
// intent to choose a document like for example a file
// file via the system's file browser.
val intent = Intent(Intent.ACTION_OPEN_DOCUMENT)

// Filter to only show results that can be "opened", such
// as a file (as opposed to a list of informational items
 )
intent.addCategory(Intent.CATEGORY_OPENABLE)

// You can use a filter to for example show only images.
// To search for all documents instead, you can use "*/*"
// here.
intent.type = "image/*"

// The actual Intent call - the system will provide the
// GUI
startActivityForResult(intent, READ_REQUEST_CODE)
```

Once an item is selected from inside the system picker, to catch the intent return you'd write something like this:

```
override fun onActivityResult(requestCode:Int,
      resultCode:Int,
      resultData:Intent) {
```

```
  // The ACTION_OPEN_DOCUMENT intent was sent with the
  // request code READ_REQUEST_CODE. If the request
  // code seen here doesn't match, it's the
  // response to some other intent, and the code below
  // shouldn't run at all.

  if (requestCode == READ_REQUEST_CODE
        && resultCode == Activity.RESULT_OK) {
      // The document selected shows up in
      // intent.getData()
      val uri = resultData.data
      Log.i("LOG", "Uri: " + uri.toString())
      showImage(uri) // Do s.th. with it
  }
}
```

Instead of opening a file as shown in the example, you can do other things with the URI you received in the intent's return. You could, for example, also issue a query to fetch metadata as shown in the previous query methods. Since the `DocumentsProvider` inherits from `ContentProvider`, you can use the methods described earlier to open a stream for the document's bytes.

Chapter 7

Permissions

Securing sensitive data is an important task during the development of apps. With more and more apps on handheld devices being used for sensitive everyday tasks such as banking, security has been gaining more importance, and it will continue to do so in the future. You as a developer must take every precaution possible to handle your app users' data responsibly.

Fully covering every possible security aspect is a challenging task and would fill a whole book on its own. Fortunately, there is a vast number of online resources you can consult to get updated with Android OS security matters. Just be cautious to filter out inappropriate information. The following security-related topics in the Android OS's online resources are a good place to start:

```
https://developer.android.com/training/best-security.html
https://developer.android.com/training/best-permissions-ids.html
```

If these links are broken when you read the book, search for *android best practices security* and *android best practices permissions* in your favorite search engine and you'll readily find these resources.

Having said that, we still want to thoroughly deal with the *permission* system inside the Android OS because this is the place you as a developer will definitely have to feel at home once your app addresses sensitive data. Permissions add security to system data and features; you can use predefined permissions, define them yourself, or declare them by writing appropriate entries in `AndroidManifest.xml`.

Permission Types

Permissions come in several flavors according to the desired protection level.

- *Normal*: This level corresponds to low-level security-sensitive information. The system will automatically grant such permissions without explicitly asking the user, but the permission is listed in the package description and can be queried by explicit demand using the system settings app.

P. Späth, *Pro Android with Kotlin,* https://doi.org/10.1007/978-1-4842-3820-2_7

- *Dangerous*: This level corresponds to high-level security-sensitive information. The user will be asked whether they want to allow using that permission. Once allowed for an app, the allowance will be saved, and the user won't be asked again until the app gets reinstalled or the permission gets explicitly revoked by using the system settings app.
- *Signature*: This level corresponds to extremely high-level security-sensitive information. Only apps signed with the same certificate as the app defining the permission can acquire it. The system will check whether the signatures match and then automatically grant the permission. This level makes sense only for a collection of apps developed by the same developer.
- *Special*: For a couple of use cases, the system grants access to certain system resources only by off-band acquisition methods. Namely, for permissions `SYSTEM_ALERT_WINDOW` and `WRITE_SETTINGS`, you have to declare them in the manifest *and* call special intents to acquire them. The intent action you have to use for `SYSTEM_ALERT_WINDOW` is `Settings.ACTION_MANAGE_OVERLAY_PERMISSION`, and the one for `WRITE_SETTINGS` is `Settings.ACTION_MANAGE_WRITE_SETTINGS`. Your app should use these two only if absolutely necessary.
- *Privileged or System Only*: These are for system image apps. You should not have to use them.

Permissions are gathered in permission groups. The idea is that once the user has accepted a permission request from permission A of group G1, another permission inquiry for another permission B of the same group G1 is not needed. From a user experience perspective, permission groups show an effect only if we are talking about Dangerous type permissions; permission groups for Normal permissions have no impact.

> **Note** The mapping of permissions to permission groups may change with future versions of Android. Your app thus should not rely on such a mapping. From a development perspective, you should just ignore permission groups, unless you define your own permissions and permission groups.

Defining Permissions

The Android OS includes a number of permissions defined by various built-in apps or the OS itself. In addition, you as a developer can define your own permissions to secure apps or parts of your apps.

As for the built-in permissions, they are defined by the system, and if your app needs one or several of them, you declare using them (see the "Permissions" section of the online text companion). The system will then decide based on the protection level what to do with these permission requests. If your app exposes sensitive information to other apps or the system and it is not handled by permissions *used* by the app, you define your own permissions inside `AndroidManifest.xml`.

```
<permission android:description="string resource"
      android:icon="drawable resource"
      android:label="string resource"
      android:name="string"
      android:permissionGroup="string"
      android:protectionLevel=["normal" | "dangerous" |
              "signature" | "signatureOrSystem"] />
```

The meaning of these attributes are described in the section “Manifest Top Level Entries” of the online text companion. At a bare minimum, you must provide the `name` and `protectionLevel` attributes, but it certainly is a good idea to also add a label, icon, and a description to help your users understand what the permission does.

If you need to group your permissions, you can use one of two methods.

- Use the `<permission-group>` element and add `permissionGroup` attributes to `<permission>`; see the section “Manifest Top Level Entries” in the online text companion.
- Use the `<permission-tree>` element and name your permissions accordingly; see the ”Manifest Top Level Entries” section in the online text companion.

If you then acquire a permission of a group, the sibling permissions from the same group will be implicitly included in the grant.

> **Caution** To adhere to security guidelines and to make your app design clear and stable, keep the number of permissions you define yourself at the bare minimum.

Using Permissions

To use permissions, inside your `AndroidManifest.xml` file add one or more, as shown here:

```
<uses-permission android:name="string"
      android:maxSdkVersion="integer" />
```

Or if you need to specify permissions for API levels 23 or higher (Android 6.0), use this:

```
<uses-permission-sdk-23 android:name="string"
      android:maxSdkVersion="integer" />
```

In both cases, the `name` attribute specifies the permission name, and `maxSdkVersion` is the maximum API level this permission requirement will take into account. This special `<uses-permission-sdk23>` element comes from a major change in permission semantics for Android 6.0. If you don’t care for that distinction, just omit the `maxSdkVersion` attribute.

The question is, how would we know which permissions exactly we need for our app? The answer has three parts.

- Android Studio tells you about a permission your app needs. If you, for example, write the following, Android Studio tells you a certain permission is required (Figure 7-1):

```
val uri = CallLog.Calls.CONTENT_URI
val cursor = contentResolver.query(
          uri, null, null, null, null)
```

- During development and testing, your app crashes and in the logs you see an entry like this:

```
Caused by: java.lang.SecurityException: Permission
Denial: opening provider
com.android.providers.contacts.CallLogProvider
from
ProcessRecord{faeda9c 4127:de.pspaeth.cp1/u0a96}
(pid=4127, uid=10096) requires
android.permission.READ_CALL_LOG or
android.permission.WRITE_CALL_LOG
```

- The list of system permissions tells you that you need a certain permission for a certain task. See Table 7-1.

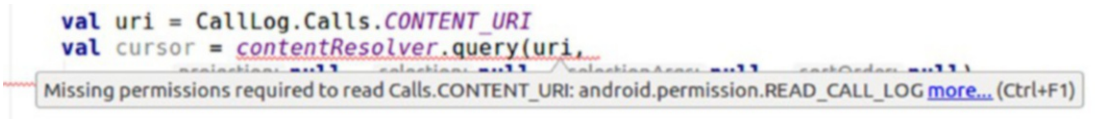

Figure 7-1. Android Studio telling you about a permission requirement

Once the usage of the permission is declared in the top-level element `<uses-permission>`, it still must be connected to the app's components. This happens either inside the `<application>` element if you want to connect the permission to all components at once or, better, on a per-component base. In either case you declare the permission inside the `permission` attribute, as follows:

```
...
<activity android:name=
        "com.example.myapp.ExampleActivity"
    android:permission=
        "com.eample.myapp.abcPermission"/>
...
```

Table 7-1. System Permissions

Permission	Group	Description
READ_CALENDAR	CALENDAR	Allows for reading the calendar. Manifest entry: android.permission.READ_CALENDAR
WRITE_CALENDAR	CALENDAR	Allows for writing the calendar. Manifest entry: android.permission.READ_CALENDAR
CAMERA	CAMERA	Allows for accessing the camera. Manifest entry: android.permission.CAMERA
READ_CONTACTS	CONTACTS	Read from the contacts table. Manifest entry: android.permission.READ_CONTACTS
WRITE_CONTACTS	CONTACTS	Write into the contacts table. Manifest entry: android.permission.WRITE_CONTACTS
GET_ACCOUNTS	CONTACTS	Allows for listing accounts from the Accounts Service. Manifest entry: android.permission.GET_ACCOUNTS
ACCESS_FINE_LOCATION	LOCATION	Allows an app to access fine-grained location. Manifest entry: android.permission.ACCESS_FINE_LOCATION
ACCESS_COARSE_LOCATION	LOCATION	Allows an app to access approximate location. Manifest entry: android.permission.ACCESS_COARSE_LOCATION
RECORD_AUDIO	MICROPHONE	Allows for recording audio. Manifest entry: android.permission.RECORD_AUDIO
READ_PHONE_STATE	PHONE	Allows read access to phone state (phone number of the device, current cellular network information, the status of any ongoing calls, and a list of any PhoneAccounts registered on the device). Manifest entry: android.permission.READ_PHONE_STATE
READ_PHONE_NUMBERS	PHONE	Read access to the device's phone number. Manifest entry: android.permission.READ_PHONE_NUMBERS
CALL_PHONE	PHONE	Allows an application to initiate a phone call without going through the dialer user interface. Manifest entry: android.permission.CALL_PONE
ANSWER_PHONE_CALLS	PHONE	Allows an application to answer an incoming phone call. Manifest entry: android.permission.ANSWER_PHONE_CALLS

(*continued*)

Table 7-1. (*continued*)

Permission	Group	Description
READ_CALL_LOG	PHONE	Allows for reading from the call log table. Manifest entry: android.permission.READ_CALL_LOG
WRITE_CALL_LOG	PHONE	Allows for writing to the call log table. Manifest entry: android.permission.WRITE_CALL_LOG
ADD_VOICEMAIL	PHONE	Allows for adding a voicemail. Manifest entry: com.android.voicemail.permission.ADD_VOICEMAIL
USE_SIP	PHONE	Allows for using the SIP service. Manifest entry: android.permission.USE_SIP
PROCESS_OUTGOING_CALLS	PHONE	Allows an application to see the number being dialed during an outgoing call with the option to redirect the call to a different number or abort the call. Manifest entry: android.permission.PROCESS_OUTGOING_CALLS
BODY_SENSORS	SENSORS	Allows an application to access data from sensors that the user uses to measure what is happening inside their body. Manifest entry: android.permission.BODY_SENSORS
SEND_SMS	SMS	Allows sending an SMS. Manifest entry: android.permission.SEND_SMS
RECEIVE_SMS	SMS	Allows receiving an SMS. Manifest entry: android.permission.RECEIVE_SMS
READ_SMS	SMS	Allows reading an SMS. Manifest entry: android.permission.READ_SMS
RECEIVE_WAP_PUSH	SMS	Allows receiving a WAP push message. Manifest entry: android.permission.RECEIVE_WAP_PUSH
RECEIVE_MMS	SMS	Allows receiving an MMS. Manifest entry: android.permission.RECEIVE_MMS
READ_EXTERNAL_STORAGE	STORAGE	Allows for reading from the external storage. Required only if the API level is below 19. Manifest entry: android.permission.READ_EXTERNAL_STORAGE
WRITE_EXTERNAL_STORAGE	STORAGE	Allows for writing to the external storage. Required only if the API level is below 19. Manifest entry: WRITE_EXTERNAL_STORAGE

Acquiring Permissions

The way permissions are handled by the Android OS has changed. Prior to Android 6.0 (API level 23), the permission inquiry asked the user happened during the installation. Starting with API level 23, a paradigm change happened: permission inquiry happens during the runtime of an app. This made the permission system more flexible; users of your app might never use certain parts of it, and thus asking for permission to do so might annoy them.

The downside of this approach is that more programming work is needed. The runtime permission inquiry must be included in your code. To do so, at any suitable place before the permission is needed, you add the following:

```
val activity = this
val perm = Manifest.permission.CAMERA
val cameraPermReturnId = 7239 // any suitable constant
val permissionCheck = ContextCompat.checkSelfPermission(
      activity, perm)
if (permissionCheck !=
          PackageManager.PERMISSION_GRANTED) {
    // Should we show an explanation?
  if (ActivityCompat.
      shouldShowRequestPermissionRationale(
            activity, perm)) {
      // Show an explanation to the user
      // *asynchronously* -- don't block
      // this thread waiting for the user's
      // response! After the user sees the
      // explanation, try again to request
      // the permission.
      val dialog = AlertDialog.Builder(activity) ...
             .create()
      dialog.show()
  } else {
      // No explanation needed, we can request
      // the permission.
      ActivityCompat.requestPermissions(activity,
            arrayOf(perm), cameraPermReturnId)
      // cameraPermReturnId is an app-defined
      // int constant. The callback method gets
      // the result of the request.
    }
}
```

This code does the following:

- First we check whether the permission has already been granted. If the permission was granted before, the user wouldn't be asked again unless the app got reinstalled or the permission got revoked explicitly.
- The `ActivityCompat.shouldShowRequestPermissionRationale()` method checks whether a rationale should be shown to the user. The idea behind that is if the user denied the permission inquiry request a couple of times, they might have done that because the need for the permission was not well understood. In this case, the app gets a chance to tell

the user more about the permission need. The frequency of how often `shouldShowRequestPermissionRationale()` returns `true` is up to the Android OS. The example here shows a dialogue; you can of course do whatever you want here to inform the user.

- The `ActivityCompat.requestPermissions(...)` method finally performs the permission inquiry. This happens asynchronously, so the call returns immediately.

Once the call to `ActivityCompat.requestPermissions(...)` happens, the user gets asked by the Android OS, outside your app, whether they want to grant the permission. The result of that will show up in an asynchronous callback method as follows:

```
override
fun onRequestPermissionsResult(
      requestCode: Int, permissions: Array<String>,
      grantResults: IntArray) {
   when (requestCode) {
     cameraPermReturnId -> {
       // If request is canceled, the result
       // arrays are empty. Here we know it just
       // can be one entry
       if ((grantResults.isNotEmpty()
             && grantResults[0] ==
                 PackageManager.PERMISSION_GRANTED)) {
         // permission was granted
         // act accordingly...
       } else {
         // permission denied
         // act accordingly...
       }
       return
    }
    // Add other 'when' lines to check for other
    // permissions this App might request.
    else -> {
      // Ignore all other requests.
      // Or whatever makes sense to you.
    }
  }
}
```

This method needs to be implemented inside an `android.content.Activity` class. In other contexts, this is not possible.

Acquiring Special Permissions

Using `ActivityCompat.requestPermissions()` in certain circumstances is not enough to acquire permissions `SYSTEM_ALERT_WINDOW` and `WRITE_SETTINGS`. For those two permissions, you need to follow a different approach.

The permission `WRITE_SETTINGS` for API levels 23 and higher must be acquired using a special intent as follows:

```
val backFromSettingPerm = 6183    // any suitable constant
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
    val activity = this
    if (!Settings.System.canWrite(activity)) {
        // This is just a suggestion: present a special
        // dialog to the user telling about the special
        // permission. Important is the Activity start
        AlertDialog dialog =
          new AlertDialog.Builder(activity)
            .setTitle(...)
            .setMessage(...)
            .setPositiveButton("OK", { dialog, id ->
                val intent = Intent(
                   Settings.ACTION_MANAGE_WRITE_SETTINGS)
                intent.data = Uri.parse("package:" +
                   getPackageName())
                 activity.startActivityForResult(intent,
                   backFromSettingPerm)
            }).setNegativeButton("Cancel",
                { dialog, id ->
                  // ...
            })
            .create();
         dialog.show();
         systemWillAsk = true;
    }
} else {
    // do as with any other permissions...
}
```

Once done with that intent, the callback method `onActivityResult()` can be used to continue with the GUI flow.

```
override
protected fun onActivityResult(requestCode:Int,
      resultCode:Int, data:Intent) {
  if ((requestCode and 0xFFFF) == backFromSettingPerm) {
      if (resultCode == Activity.RESULT_OK) {
         // act accordingly...
      }
  }
}
```

For the `SYSTEM_ALERT_WINDOW` permission, you have to follow the same approach, but use `ACTION_MANAGE_OVERLAY_PERMISSION` instead for creating the content.

> **Note** For this special `SYSTEM_ALERT_WINDOW` permission, the Google Play store will automatically grant the permission if the app gets installed from the Google Play store and the API level is 23 or higher. For local development and testing, you have to use the intent as described.

Feature Requirements and Permissions

In Chapter 2 we saw that by virtue of the <uses-feature> element inside AndroidManifest.xml you can specify which features your app will use. This information is important for the Google Play store to find out on which devices your app can run after published. However, there is another important aspect to take into account if you specify this requirement: which permissions will be implied by such requirements, and how will they be handled depending on the API level in use?

Feature constants and API levels do not necessarily strictly relate to each other. For example, the android.hardware.bluetooth feature was added in API level 8, but the corresponding Bluetooth API was added in API level 5. Because of this, some apps were able to use the API before they had the ability to declare that they require the API using the <uses-feature> declaration. To remedy this discrepancy, Google Play assumes that certain hardware-related permissions indicate that the underlying hardware features are required by default. For instance, applications that use Bluetooth must request the BLUETOOTH permission in a <uses-permission> element, and for apps targeting older API levels, Google Play assumes that the permission declaration implies that the underlying android.hardware.bluetooth feature is required by the application. Table 7-2 lists the permissions that imply such feature requirements.

Note that the <uses-feature> declarations take precedence over features implied by the permissions in Table 7-2. For any of these permissions, you can disable filtering based on the implied feature by explicitly declaring the implied feature in a <uses-feature> element, with an android:required="false" attribute. For example, to disable any filtering based on the CAMERA permission, you would add this to the manifest file:

```
<uses-feature android:name="android.hardware.camera"
              android:required="false" />
```

Table 7-2. *Permissions That Imply Feature Requirements*

Category	Permission...	...Implies Feature
Bluetooth	BLUETOOTH	android.hardware.bluetooth
	BLUETOOTH_ADMIN	android.hardware.bluetooth
Camera	CAMERA	android.hardware.camera and android.hardware.camera.autofocus
Location	ACCESS_MOCK_LOCATION	android.hardware.location
	ACCESS_LOCATION_EXTRA_COMMANDS	android.hardware.location
	INSTALL_LOCATION_PROVIDER	android.hardware.location
	ACCESS_COARSE_LOCATION	android.hardware.locationandroid.hardware.location.network (API level < 21)
	ACCESS_FINE_LOCATION	android.hardware.locationandroid.hardware.location.gps (API level < 21)
Microphone	RECORD_AUDIO	android.hardware.microphone

(*continued*)

Table 7-2. (*continued*)

Category	Permission...	...Implies Feature
Telephony	CALL_PHONE	android.hardware.telephony
	CALL_PRIVILEGED	android.hardware.telephony
	MODIFY_PHONE_STATE	android.hardware.telephony
	PROCESS_OUTGOING_CALLS	android.hardware.telephony
	READ_SMS	android.hardware.telephony
	RECEIVE_SMS	android.hardware.telephony
	RECEIVE_MMS	android.hardware.telephony
	RECEIVE_WAP_PUSH	android.hardware.telephony
	SEND_SMS	android.hardware.telephony
	WRITE_APN_SETTINGS	android.hardware.telephony
	WRITE_SMS	android.hardware.telephony
Wi-Fi	ACCESS_WIFI_STATE	android.hardware.wifi
	CHANGE_WIFI_STATE	android.hardware.wifi
	CHANGE_WIFI_MULTICAST_STATE	android.hardware.wifi

Permissions Handling Using a Terminal

To see the permissions you have registered on your device, you can scan through the apps list in the system settings app or, more easily, use the ADB shell to get various permission-related information in a terminal.

For that aim, connect the hardware-device via USB to your laptop or PC, open a terminal, `cd` to the `platform-tools` folder in your SDK installation, find your device in `./adb devices`, and then enter the following:

```
./adb shell -s <DEVICE-NAME>
```

If there is only one device, you can omit that `-s` switch.

Once inside the shell, you can use a couple of commands to get permission information. First you can list all packages installed via this:

```
cmd package list package
```

To show all Dangerous permissions, to see the permission state for a certain package, or to grant or revoke one or more permissions, you can use the following:

```
cmd package list permissions -d -g
dumpsys package <PACKAGE-NAME>
pm [grant|revoke] <PERMISSION-NAME> ...
```

> **Note** Current versions of `dumpsys` will show both requested *and* granted permissions. Do not get confused by old blog entries about that matter.

Chapter 8

APIs

The subject of this chapter is to introduce APIs, which are the cornerstones of your app. The APIs include the following:

- Databases
- Scheduling
- Loaders
- Notifications
- Alarm Manager
- Contacts
- Search Framework
- Location and Maps

Databases

Android provides two realms for dealing with databases: either you use the SQLite library included in the Android OS, or you use the Room architecture component. The latter is recommended since it adds an abstraction layer between the database and the client, simplifying the mapping between Kotlin objects and database storage objects. You can find exhaustive information about SQLite in the online docs and lots of examples on the Web. In this book, we talk about Room since the separation of concerns induced by the abstraction helps you to write better code. Also, since Room helps to avoid boilerplate code, you can shorten your database code significantly if you use Room instead of SQLite.

P. Späth, *Pro Android with Kotlin,* https://doi.org/10.1007/978-1-4842-3820-2_8

Configuring Your Environment for Room

Since Room is a support architecture component, you must configure it in your Android Studio build script. To do so, open the module's `build.gradle` file (not the one from the project!) and on top level (not inside any of the curly braces) write the following:

```
apply plugin: 'kotlin-kapt'
```

This is the Kotlin compiler plugin that supports annotation processing. In the `dependencies` section, write the following (on three lines; remove the newlines after `implementation` and `kapt`):

```
// Room
implementation
    "android.arch.persistence.room:runtime:1.0.0"
kapt
    "android.arch.persistence.room:compiler:1.0.0"
```

Room Architecture

Room was designed with ease of use in mind; you basically deal with three kinds of objects.

- *Database*: Represents a holder for the database. Talking in SQL language idioms, it contains several tables. To say it in a technology-agnostic way, a database contains several entity containers.
- *Entity*: Represents a table in the SQL world. To say it in a technology-agnostic way, this is a usage-centric aggregate of fields. An example would be an employee inside a company or a contact holding information about how to communicate with people or partners.
- *Data Access Object (DAO)*: Contains the access logic to retrieve data from the database. It thus serves as an interface between the program logic and the database model. You often have one DAO per entity class but possibly more DAOs for various combinations. You could, for example, have an `EmployeeDao` and a `ContactDao` for the two employee and contact entities, as well as a `PersonDao` that combines the employee and the contacts information of a person.

The Database

To declare a database, you write the following:

```
import android.arch.persistence.room.*

@Database(entities =
    arrayOf(Employee::class, Contact::class),
    version = 1)
abstract class MyDatabase : RoomDatabase() {
    abstract fun employeeDao(): EmployeeDao
```

```
    abstract fun contactDao(): ContactDao
    abstract fun personDao(): PersonDao
}
```

Inside the `@Database` annotation you declare all the entity classes used, and as abstract functions you provide factory methods for the DAO classes. You don't have to implement this abstract database class. The Room library will automatically provide an implementation for you based on the signatures and the annotations! The version number will help you when upgrading to different data model versions; you'll learn more about that in the following sections.

Entities

Next we implement the entity classes, which is extremely easy to do in Kotlin.

```
@Entity
data class Employee(
    @PrimaryKey(autoGenerate = true) var uid:Int = 0,
    var firstName:String,
    var lastName:String)

@Entity
data class Contact(
    @PrimaryKey(autoGenerate = true) var uid:Int = 0,
    var emailAddr:String)
```

You can see that we need a primary key of type `Int` for each entity. `autoGenerate = true` takes care of automatically making it unique.

The column names from the database table defined by these entity classes match the variable names. If you want to change that, you can add another annotation:

```
@ColumnInfo.@Entity
data class Employee(
    @PrimaryKey(autoGenerate = true) var uid:Int = 0,
    @ColumnInfo(name = "first_name") var firstName:String,
    @ColumnInfo(name = "last_name") var lastName:String)
```

This would lead to using `first_name` and `last_name` as table column names.

Also, the table name is taken from the entity class name, like with `Employee` and `Contact` for these examples. You can also change this; just add the parameter `tableName` to the `@Entity` annotation as follows:

```
@Entity(tableName = "empl")
data class Employee(
    @PrimaryKey(autoGenerate = true) var uid:Int = 0,
    @ColumnInfo(name = "first_name") var firstName:String,
    @ColumnInfo(name = "last_name") var lastName:String)
```

While it is generally a good idea to have a single integer-valued primary key, you can also use a combined key. For that aim, there is an additional annotation parameter in `@Entity`. Here's an example:

```
@Entity(tableName = "empl",
        primaryKeys = tableOf("first_name","last_name"))
data class Employee(
    @ColumnInfo(name = "first_name") var firstName:String,
    @ColumnInfo(name = "last_name") var lastName:String)
```

Entities can also have fields that will not be persisted. From a design perspective, this is maybe not a good idea, but if you need such a field, you can add it and use the annotation `@Ignore` as follows:

```
@Entity(tableName = "empl")
data class Employee(
    @PrimaryKey(autoGenerate = true) var uid:Int = 0,
    var firstName:String = "",
    var lastName:String = "",
    @Ignore var salary:Int)
```

Because of the way Room is implemented, if you add such an `@Ignore` annotation, *all* the fields must have default values assigned, even if unused.

Relationships

Room by design doesn't allow direct relationships between entities. You cannot, for example, add a list of `Contact` entities as a class member of an `Employee` entity. However, it is possible to declare foreign key relationships, which helps in maintaining data consistency.

To do so, add a `foreignKeys` annotation attribute, as in the following code snippet:

```
@Entity(
        foreignKeys = arrayOf(
            ForeignKey(entity = Employee::class,
                 parentColumns = arrayOf( "uid" ),
                 childColumns = arrayOf( "employeeId" ),
                 onDelete = ForeignKey.CASCADE,
                 onUpdate = ForeignKey.CASCADE,
                 deferred = true)),
        indices = arrayOf(
             Index("employeeId"))
)
@Entity
data class Contact(
        @PrimaryKey(autoGenerate = true) var uid:Int = 0,
        var employeeId:Int,
        var emailAddr:String)
```

Here are a few notes about this construct:

- In Java you would write `@Entity(foreignKeys = @ForeignKey( ... )`. Kotlin doesn't allow annotations inside annotations. In this case, using the constructor serves as a substitute, which boils down to omitting the `@` for inner annotations.
- In a Java annotation, attribute value arrays are written like `name = { ..., ... }`. This cannot be used in Kotlin because the curly braces do not serve as array initializers. Instead, the `arrayOf(...)` library method gets used.
- The `childColumns` attribute points to the reference key in *this* entity, `Contact.employeeId` in this case.
- The `parentColumns` attribute points to the referred-to foreign key entity, in this case `Employee.uid`.
- The `onDelete` attribute tells what to do if the parent gets deleted. A value of `ForeignKey.CASCADE` means to also automatically remove all children, which is the associated `Contact` entities. The possible values are as follows:
  - `CASCADE`: Transport all actions to the root of the child-parent relation tree.
  - `NO_ACTION`: Don't do anything. This is the default, and it leads to an exception if the relationship breaks because of update or delete actions.
  - `RESTRICT`: Similar to `NO_ACTION`, but the check will be made immediately when a delete or an update happens.
  - `SET_NULL`: All child key columns get set to `null` if a parent delete or update happens.
  - `SET_DEFAULT`: All child key columns get set to their default if a parent delete or update happens.
- The `onUpdate` attribute tells what to do if the parent gets updated. A value `ForeignKey.CASCADE` means to also automatically update all children, which are the associated `Contact` entities. The possible values are the same as for `onDelete`.
- The `deferred = true` setting will postpone the consistency check until the database transaction is committed. This might, for example, be important if both parent and child get created inside the same transaction.
- Foreign keys must be part of a corresponding index. Here `Contact.employeeId` gets the index. You'll learn more about indexes in the following sections.

Nested Objects

Although it is not possible to define inter-object relations other than manually by foreign keys, you can on the object side define a nesting of hierarchical objects. For example, from the following employee entity:

```
@Entity
data class Employee(
    @PrimaryKey(autoGenerate = true) var uid:Int = 0,
    var firstName:String,
    var lastName:String)
```

you can factor out the first and last names and instead write the following:

```
data class Name(var firstName:String, var lastName:String)

@Entity
data class Employee(
    @PrimaryKey(autoGenerate = true) var uid:Int = 0,
    @Embedded var name:Name)
```

Note that this does not have any impact on the database side of the data model. The associated table will still have the columns `uid`, `firstName`, and `lastName`. Since the database identity of such an embedded object is tied to the name of its fields, if you have several embedded objects of the same embedded type, you must disambiguate the names by using a `prefix` attribute as follows:

```
data class Name(var firstName:String, var lastName:String)

@Entity
data class Employee(
    @PrimaryKey(autoGenerate = true) var uid:Int = 0,
    @Embedded var name:Name,
    @Embedded(prefix="spouse_") var spouseName:Name)
```

This makes the table have the columns `uid`, `firstName`, `lastName`, `spouse_firstName`, and `spouse_lastName`.

If you like, inside the embeddable class, you can use Room annotations. For example, you can use the `@ColumnInfo` annotation to specify custom column names.

```
data class Name(
  @ColumnInfo(name = "first_name") var firstName:String,
  @ColumnInfo(name = "last_name") var lastName:String)

@Entity
data class Employee(
    @PrimaryKey(autoGenerate = true) var uid:Int = 0,
    @Embedded var name:Name)
```

Using Indexes

To improve database query performance, you can declare one or more indexes to use on certain fields or field combinations. You don't have to do that for the unique key; this is done automatically for you. But for any other index you want to define, write something like this:

```
@Entity(indices = arrayOf(
    Index("employeeId"),
    Index(value = arrayOf("country","city"))
    )
)
data class Contact(
        @PrimaryKey(autoGenerate = true) var uid:Int = 0,
        var employeeId:Int,
        var emailAddr:String,
        var country:String,
        var city:String)
```

This adds an index that allows for fast queries using the foreign key field `employeeId` and another one for fast queries given both country and city.

If you add `unique = true` as an attribute to the `@Index` annotation, Room will make sure the table cannot have two entries with the same value for that particular index. As an example, we can add a Social Security number (SSN) field to `Employee` and define an unique index for it, as shown here:

```
@Entity(indices = arrayOf(
    Index(value = arrayOf("ssn"), unique = true)
    )
)
data class Employee(
    @PrimaryKey(autoGenerate = true) var uid:Int = 0,
    var ssn:String,
    @Embedded var name:Name)
```

If you now try to add two employees with the same SSN to the database, Room will throw an exception.

Data Access: DAOs

Data access objects (DAOs) provide the logic to access the database. We have already seen that inside the database declaration we had to list all DAOs in factory methods as follows:

```
@Database(entities =
    arrayOf(Employee::class, Contact::class),
    version = 1)
abstract class MyDatabase : RoomDatabase() {
    abstract fun employeeDao(): EmployeeDao
    abstract fun contactDao(): ContactDao
    abstract fun personDao(): PersonDao
}
```

In this example, we declare three DAOs for use by Room. For the actual implementation, we don't need fully fledged DAO classes. It is enough to declare interfaces or abstract classes, and Room will do the rest for us.

The DAO classes, for example, for the following entity:

```
@Entity
data class Employee(
    @PrimaryKey(autoGenerate = true) var uid:Int = 0,
    @ColumnInfo(name = "first_name") var firstName:String,
    @ColumnInfo(name = "last_name") var lastName:String)
```

might look like this:

```
@Dao
interface EmployeeDao {
    @Query("SELECT * FROM employee")
    fun getAll(): List<Employee>

    @Query("SELECT * FROM employee" +
        " WHERE uid IN (:uIds)")
    fun loadAllByIds(uIds: IntArray): List<Employee>

    @Query("SELECT * FROM employee" +
        " WHERE last_name LIKE :name")
    fun findByLastName(name: String): List<Employee>

    @Query("SELECT * FROM employee" +
        " WHERE last_name LIKE :lname AND " +
        "       first_name LIKE :fname LIMIT 1")
    fun findByName(lname: String, fname: String): Employee

    @Query("SELECT * FROM employee" +
        " WHERE uid = :uid")
    fun findById(uid: Int): Employee

    @Insert
    fun insert(vararg employees: Employee): LongArray

    @Update
    fun update(vararg employees: Employee)

    @Delete
    fun delete(vararg employees: Employee)
}
```

You see that we used an interface here, which is possible because the complete access logic is defined by method signatures and annotations. Also, for insert, update, and delete, the method signature is all that Room needs; it will send the right commands to the database just by looking at the signatures.

For the various query methods, we use `@Query` annotations to provide the correct database commands. You can see that Room is smart enough to see whether we want to return a list of objects or a single object. Also, we can pass method arguments into the pseudo-SQL by using `:name` identifiers.

The `@Insert` annotation allows for adding the attribute `onConflict = "<strategy>"` where you can specify what to do if a conflict occurs because a unique or primary key constraint is violated. Possible values for the `<strategy>` are given inside constants:

- `OnConflictStrategy.ABORT` to abort the transaction
- `OnConflictStrategy.FAIL` to fail the transaction
- `OnConflictStrategy.IGNORE` to ignore the conflict
- `OnConflictStrategy.REPLACE` to just replace the entity and otherwise continue the transaction
- `OnConflictStrategy.ROLLBACK` to roll back the transaction

The other DAOs from the example entities used earlier will look similar. `PersonDao` might do outer joins to combine the employee and contact entities:

```
@Dao
interface ContactDao {
    @Insert
    fun insert(vararg contacts: Contact)

    @Query("SELECT * FROM Contact WHERE uid = :uId")
    fun findById(uId: Int): List<Contact>

    @Query("SELECT * FROM Contact WHERE" +
           " employeeId = :employeeId")
    fun loadByEmployeeId(employeeId: Int): List<Contact>
}

data class Person(@Embedded var name:Name?,
                  var emailAddr: String?)
@Dao
interface PersonDao {
    @Query("SELECT * FROM empl" +
           " LEFT OUTER JOIN Contact ON" +
           "     empl.uid = Contact.employeeId" +
           " WHERE empl.uid = :uId")
    fun findById(uId: Int): List<Person>
}
```

Observable Queries

In addition to performing a query with returning entities or lists or arrays of entities as they are at the moment when the query happens, it is also possible to retrieve the query result *plus* register an observer that gets invoked when the underlying data change.

The construct to achieve this for a method inside a DAO class looks like this:

```
@Query("SELECT * FROM employee")
    fun getAllSync(): LiveData<List<Employee>>
```

So, you basically wrap a `LiveData` class around the result, and this is what you can do with all your queries.

However, this is possible only if you add the corresponding architecture component. For this aim, add the following to your module's `build.gradle` file:

```
implementation "android.arch.lifecycle:livedata:1.1.0"
```

This `LiveData` object now allows for adding an observer as follows:

```
val ld: LiveData<List<Employee>> =
      employeeDao.getAllSync()
ld.observeForever { l ->
    l?.forEach { empl ->
      Log.e("LOG", empl.toString())
      // do s.th. else with the employee
    }
}
```

This is particularly useful if inside the observer callback you update GUI components.

> **Caution** Your production code should do a better job in doing correct housekeeping. The `LiveData` object should have the observer unregistered by calling `ld.removeObserver(...)` at an appropriate place in your code. It is not shown here because we just provide snippets, and the housekeeping must be done in the code *containing* the snippets.

A `LiveData` object also allows for adding an observer tied to a lifecycle object. This is done with the following:

```
val ld: LiveData<List<Employee>> =
      employeeDao.getAllSync()
val lcOwn : LifecycleOwner = ...
ld.observe(lcOwn, { l ->
    l?.forEach { empl ->
      Log.e("LOG", empl.toString())
      // do s.th. else with the employee
    }
} )
```

For details about lifecycle objects, please take a look at the online API documentation for `android.arch.lifecycle.LiveData`.

A similar but maybe more comprehensive approach to add observables to your database code is to use RxJava/RxKotlin, which is the Java/Kotlin platform implementation of ReactiveX. We do not give an introduction to ReactiveX programming here, but including it in queries boils down to wrapping the results into RxJava objects. To give you a quick idea of how to do that, you, for example, write the following:

```
@Query("SELECT * FROM employee" +
    " WHERE uid = :uid")
fun findByIdRx(uid: Int): Flowable<Employee> {
   [...] // Wrap query results into a Flowable
}
```

This returns a `Flowable`, allowing *observers* to *react* on retrieved database rows in an asynchronous manner.

For this to work, you have to include RxJava support into the build file (remove the newline after `implementation`).

```
// RxJava support for Room
Implementation
    "android.arch.persistence.room:rxjava2:1.0.0"
```

For more details about RxKotlin, please consult the online resources about ReactiveX in general or RxKotlin for the Kotlin language binding of ReactiveX.

Database Clients

To actually include Room into the app, we need to know how we can get hold of databases and DAO objects. To achieve this, we first acquire a reference to a database via the following:

```
fun fetchDb() =
    Room.databaseBuilder(
        this, MyDatabase::class.java,
        "MyDatabase.db")
    .build()
val db = fetchDb()
```

This creates a file-backed database. The string argument is the name of the file holding the data. To instead open a memory-based database, say for testing purposes or because you favor speed over data loss when the application stops, use the following:

```
fun fetchDb() =
    Room.inMemoryDatabaseBuilder(
        this, MyDatabase::class.java)
    .build()
val db = fetchDb()
```

The builder allows for certain configuration activities in a fluent builder style. Interesting configuration options are shown in Table 8-1. You just chain them before the final `.build()` call. One option you might use often during early development phases is relaxing the foreground operation restriction by using this:

```
fun fetchDb() =
    Room.databaseBuilder(
        this, MyDatabase::class.java,
        "MyDatabase.db")
    .allowMainThreadQueries()
    .build()
val db = fetchDb()
```

***Table 8-1.** Room Builder Options*

Option	Description
`addCallback(RoomDatabase.Callback)`	Use this to add a `RoomDatabase.Callback` to this database. You can use it, for example, to have some code executed when the database gets created or opened.
`allowMainThreadQueries()`	Use this to disable the no main thread restriction in Room. If you don't use this and try to perform database operations in the main thread, Room will throw an exception. There is a good reason for Room to work this way. GUI-related threads should not be blocked because of lengthy database operations. So, for your code you should not call this method; it makes sense only for experiments to avoid dealing with asynchronicity.
`addMigrations(vararg Migration)`	Use this to add migration plans. Migration is covered in more detail later in the chapter.
`fallbackToDestructiveMigration()`	If a matching migration plan is missing (for example, for a necessary upgrade from the data version *inside* the database to the version specified in the `@Database` annotation, no registered migration plan can be found), Room normally throws an exception. If you instead want the current database to be purged and then the database be built up from scratch for the new version, use this method.
`fallbackToDestructive Migration(vararg Int)`	This is the same as `fallbackToDestructiveMigration()` but restricted to certain starting versions. For all other versions, an exception will be thrown if the migration plan is missing.

Then, once you have a database object, just call any of the DAO factory methods we defined inside the database class in an abstract manner, and Room automatically provides implementations. So, for example, write the following:

```
val db = ...
val employeeDao = db.employeeDao()
// use the DAO...
```

Transactions

Room allows for transactions in `EXCLUSIVE` mode. This means that if transaction A is in progress, no other processes or threads are allowed to access a database in another transaction B until transaction A is finished. More precisely, transaction B will have to wait until A is finished.

To run a set of database operations inside a transaction in Kotlin, you can write the following:

```
val db = ...
db.runInTransaction { ->
    // do DB work...
}
```

The transaction is marked ”successful” if the code inside the closure does not throw any exception. Otherwise, the transaction will be rolled back.

Migrating Databases

To migrate databases from one version of your app to another version, you add migration plans while accessing the database as follows:

```
val migs = arrayOf(
  object : Migration(1,2) {
    override fun migrate(db: SupportSQLiteDatabase) {
      // code for the 1->2 migration...
      // this is already running inside a transaction,
      // don't add your own transaction code here!
    }
  }, object : Migration(2,3) {
    override fun migrate(db: SupportSQLiteDatabase) {
      // code for the 2->3 migration...
      // this is already running inside a transaction,
      // don't add your own transaction code here!
    }
  } // more migrations ...
    )

    private fun fetchDb() =
        Room.databaseBuilder(
              this, MyDatabase::class.java,
              "MyDatabase.db")
          .addMigrations(*migs)
          .build()
```

It obviously makes no sense to use DAO classes here, because then you’d have to manage several DAO variants, one for each version. That is why inside the `migrate()` methods you need to access the database on a lower level, for example by executing SQL statements without bindings to Kotlin objects. As an example, say you have an `Employee` table. You upgrade from version 1 to 2 and need to add a column `salary`, and then you upgrade

from version 2 to 3 and need another column, `childCount`. Inside the `migs` array from the previous code, you then write the following:

```
//...
object : Migration(1,2) {
  override fun migrate(db: SupportSQLiteDatabase) {
    db.execSQL("ALTER TABLE components "+
           "ADD COLUMN salary INTEGER DEFAULT 0;")
  }
}
//...
object : Migration(2,3) {
  override fun migrate(db: SupportSQLiteDatabase) {
    db.execSQL("ALTER TABLE components "+
           "ADD COLUMN childCount INTEGER DEFAULT 0;")
  }
}
//...
object : Migration(1,3) {
  override fun migrate(db: SupportSQLiteDatabase) {
    db.execSQL("ALTER TABLE components "+
           "ADD COLUMN salary INTEGER DEFAULT 0;")
    db.execSQL("ALTER TABLE components "+
           "ADD COLUMN childCount INTEGER DEFAULT 0;")
  }
}
//...
```

If you provide small-step migrations as well as large-step migrations, the latter will have precedence. This means if you have migration plans 1 → 2, 2 → 3, and 1 → 3 and the system demands a migration 1 → 3, the plan 1 → 3 will run, not the chain 1 → 2 → 3.

Scheduling

With user experience in mind, running tasks in an asynchronous manner is an important matter. It is vital that no lengthy operations disturb the front-end flow, leaving the impression that your app is doing its job fluently.

It is not too easy to write stable apps that have important parts running in background, though. The reasons for that are many: the device might get powered off on demand or because of low battery, or the user might have started a more important app with higher priority, expecting to temporarily run background jobs in a low-priority mode. Also, the Android OS might decide to interrupt or postpone background jobs for other reasons such as a resource shortage or because a timeout condition applies. And with the advent of Android 8, it has become even more important to think about clever ways of performing background tasks since this version imposes severe restrictions on the background execution of program parts.

For running jobs in an asynchronous manner, several techniques exist, all of them with downsides and advantages.

- **Java threads**

  Java and Kotlin threads (remember, both are targeting the same Java virtual machine) are a low-level technique of running things in the background. In Kotlin you can use a construct as easy as this to process program parts in a background thread:

  ```
  Thread{-> do s.th.}.start()
  ```

  This is a basic approach, and you can expect high performance from your background execution tasks. However, you are completely running out of any Android OS component lifecycle, so you do not really have good control of what happens to long-running background threads while the lifecycle status of Android processes changes.

- **Java concurrency classes**

  Java and Kotlin allow the use of concurrency-related classes from the `java.util.concurrency` package. This is a higher-level approach of running things in the background with improved background tasks management capabilities, but it still has the downside of running beyond the control of the Android component lifecycle.

- **AlarmManager**

  This was originally designed for running tasks at specific times, and you can use it if you need to send notifications to the user at specific instances in time. It has been there since API level 1. Starting at API level 19 (Android 4.4), the system allows for postponing alarms under certain conditions. The downside is you don't have control over more general device conditions; when the device is up, it will fire alarm events at its own discretion, no matter what else happens on your device.

- **SyncAdapter**

  This methodology was added in Android API level 5. It is particularly useful for synchronization tasks. For more general background execution tasks, you should instead use one of the following two, Firebase JobDispatcher or JobScheduler. Use one of these only if you need one of the additional functionalities it provides.

- **Firebase JobDispatcher**

  This is a general multipurpose job dispatcher library you can use for an Android device starting from API level 14, targeting more than 99 percent of the Android devices in use. It is a little hard to find comprehensive and complete documentation for the Firebase JobDispatcher on the Web, but you will find enough examples to get you started. It is not part of the Android OS core, though, and it requires Google Play Services and the Google Play store to be installed.

- **JobScheduler**

  This is an integrated library for scheduling jobs on the Android OS. It runs on any device starting at API level 21, which is for about 85 percent of the Android devices in use. It is highly recommended to use it, unless you really need to address devices before API level 21, that is, Android 4.4 and older.

The more low-level approaches are covered in Chapter 10; the rest of this section is about the Firebase JobDispatcher, the JobScheduler, and the AlarmManager.

JobScheduler

The JobScheduler is the dedicated method to schedule and run background tasks in any Android device starting at API level 21. The documentation of Android 8 strongly recommends using JobSchedulers to overcome the background task execution restrictions imposed since Android 8.

> **Note** Use JobSchedulers for background tasks if your target API level is 21 or higher (great than 85 percent of Android installations as of February 2018).

To start using a JobScheduler, we first implement the job itself. To do so, implement the class `android.app.job.JobService`, as follows:

```
class MyJob : JobService() {
    var jobThread:Thread? = null

    override
    fun onStartJob(params: JobParameters) : Boolean {
        Log.i("LOG", "MyJob: onStartJob() : " +
              params.jobId)

        jobThread?.interrupt()
        jobThread = Thread {
            Log.i("LOG", "started job thread")
            // do job work...
            jobFinished(params, false) jobThread = null
            Log.i("LOG", "finished job thread")
        }
        jobThread.start()
        return true
    }

    override
    fun onStopJob(params:JobParameters) : Boolean {
        Log.i("LOG", "MyJob: onStopJob()")
        jobThread?.interrupt()
```

```
        jobThread = null
        return true
    }
}
```

The most important part of the implementation is the `onStartJob()` method. There you'll enter the work the job is actually supposed to do. Note that we pushed the actual work into a thread. This is important because the `onStartJob()` method runs in the app's main thread, thus blocking potentially important other work if it stays too long inside. Starting a thread instead finishes immediately. Also, we return `true`, signaling that the job continues doing its work in a background thread. Once the job finishes, it must call `jobFinished()`; otherwise, the system wouldn't know that the job finished doing its work.

The overridden `onStopJob()` method is *not* part of the normal job lifecycle. It instead gets called when the system decides to finish the job prematurely. We let it return `true` to tell the system that it is allowed to reschedule the job, in case it was configured accordingly.

To finish the job implementation, we must still configure the service class inside `AndroidManifest.xml`. To do so, add the following:

```
<service android:name=".MyJob"
         android:label="MyJob Service"
         android:permission=
               "android.permission.BIND_JOB_SERVICE" />
```

The permission configured here is *not* a "dangerous" permission, so you don't have to implement a process to acquire this permission. However, you must add this permission here; otherwise, the job gets ignored.

To actually schedule a job governed by the JobScheduler, you first need to obtain a `JobScheduler` object as a system service. Then you can build a `JobInfo` object, and in the end you register it with the JobScheduler.

```
val jsched = getSystemService(JobScheduler::class.java)
val JOB_ID : Int = 7766

val service = ComponentName(this, MyJob::class.java)
val builder = JobInfo.Builder(JOB_ID, service)
    .setMinimumLatency((1 * 1000).toLong())
          // wait at least 1 sec
    .setOverrideDeadline((3 * 1000).toLong())
          // maximum delay 3 secs

jsched.schedule(builder.build())
```

This example schedules the job to be started, the earliest after one second and the latest after three seconds. By construction it gets the ID `7766` assigned. This is a value passed to `onStartJob()` inside the job implementation. The number is just an example; you can use any unique number for the ID.

While building the `JobInfo` object, you can set various job characteristics, as shown in Table 8-2.

Table 8-2. JobInfo Builder Options

Method	Description
`setMinimumLatency (minLatencyMillis: Long)`	This job should be delayed by the specified amount of time, or longer.
`setOverrideDeadline(maxExecution DelayMillis: Long)`	This is the maximum time a job can be delayed.
`setPeriodic(intervalMillis: Long)`	This makes the job repeating and sets a recurrence interval. The actual interval can be higher but will not be lower.
`setPeriodic(intervalMillis: Long, flexMillis: Long)`	This makes the job repeating and sets a recurrence interval and a flexibility window. So, the real interval will be between *intervalMillis* – 0.5 *flexMillis* and *intervalMillis* + 0.5 *flexMillis*. Both numbers get their lowest possible value clamped to `getMinPeriodMillis()` and `MAX( getMinFlexMillis(), 0.05 * intervalMillis )`, respectively.
`setBackoffCriteria(initial BackoffMillis:Long, backoffPolicy:Int)`	A back-off might happen when inside your job implementation you write `jobFinished(params, true)`. Here you specify what happens in such a case. Possible values for `backoffPolicy` are given by the constants in the following: • `JobInfo.BACKOFF_POLICY_LINEAR`: Back-offs happen at intervals of *initialBackoffMillis retry – number*. • `JobInfo.BACKOFF_POLICY_EXPONENTIAL`: Backoffs happen at intervals of *initialBackoffMillis* $2^{retry-number}$.
`setExtras(extras: PersistableBundle)`	This sets optional extras. These extras get passed to `onStartJob()` inside the job implementation.
`setTransientExtras(extras: Bundle)`	This is only for API level 26 and higher. This sets optional unpersisted extras. These extras get passed to `onStartJob()` inside the job implementation.
`setPersisted(isPersisted: Boolean)`	This sets whether the job gets persisted across device reboots. It needs the permission `android.Manifest.permission.RECEIVE_BOOT_COMPLETED`.
`setRequiredNetworkType(networkType: Int)`	This specifies an additional condition that needs to be met for the job to run. These are possible argument values: • `JobInfo.NETWORK_TYPE_NONE` • `JobInfo.NETWORK_TYPE_ANY` • `JobInfo.NETWORK_TYPE_UNMETERED` • `JobInfo.NETWORK_TYPE_NOT_ROAMING` • `JobInfo.NETWORK_TYPE_METERED`

(*continued*)

Table 8-2. (*continued*)

Method	Description
`setRequiresBatteryNotLow (batteryNotLow: Boolean)`	This is only for API level 26 and higher. This specifies as an additional condition that needs to be met for the job to run that the battery must not be low. `false` resets this to not care.
`setRequiresCharging (requiresCharging: Boolean)`	Specifies as an additional condition that needs to be met for the job to run that the device must be plugged in. `false` resets this to not care.
`setRequiresDeviceIdle (requiresDeviceIdle: Boolean)`	Specifies as an additional condition that needs to be met for the job to run that the device must be in idle state. `false` resets this to not care.
`setRequiresStorageNotLow (storageNotLow: Boolean)`	This is only for API level 26 and higher. This specifies as an additional condition that needs to be met for the job to run that the device memory must not be low. `false` resets this to not care.
`addTriggerContentUri (uri: JobInfo.TriggerContentUri)`	This is only for API level 24 and higher. This adds a content URI that will be monitored for changes. If a change happens, the job gets executed.
`setTriggerContentUpdateDelay (durationMs: Long)`	This is only for API level 24 and higher. This sets the minimum delay in milliseconds from when a content change is detected until the job is scheduled.
`setTriggerContentMaxDelay (durationMs: Long)`	This is only for API level 24 and higher. This sets the maximum total delay in milliseconds that is allowed from the first time a content change is detected until the job is scheduled.

`setClipData (clip:ClipData, grantFlags:Int)`	This is only for API level 26 and higher. This sets a `ClipData` associated with this job. Possible values for the `grantFlags` are as follows:
	`FLAG_GRANT_READ_URI_PERMISSION`
	`FLAG_GRANT_WRITE_URI_PERMISSION`
	`FLAG_GRANT_PREFIX_URI_PERMISSION`
	(All constants are inside class `Intent`.)

Firebase JobDispatcher

The Firebase JobDispatcher is an alternative to the JobScheduler that works for Android API levels before and starting with 21.

> **Caution** The Firebase JobDispatcher library requires Google Play Services and the Google Play store to be installed. If you are not targeting API levels below 21, it is recommended you use the JobScheduler instead.

To use the Firebase JobDispatcher, it first must be installed. To do so, add the following to your module's `build.gradle` file, in the `dependencies` section:

```
implementation 'com.firebase:firebase-jobdispatcher:0.8.5'
```

As a first step, implement a job class as follows:

```
import com.firebase.jobdispatcher.*

class MyJobService : JobService() {
    var jobThread:Thread? = null

    override fun onStopJob(job: JobParameters?): Boolean {
        Log.e("LOG", "onStopJob()")
        jobThread?.interrupt()
        jobThread = null
        return false // this job should not be retried
    }

    override fun onStartJob(job: JobParameters): Boolean {
       Log.e("LOG", "onStartJob()")

        jobThread?.interrupt()
        jobThread = Thread {
            Log.i("LOG", "started job thread")
            // do job work...
            jobFinished(job, false)
            // instead use true to signal a retry
            jobThread = null
            Log.i("LOG", "finished job thread")
        }
        jobThread?.start()

        return true // work is going on in the background
    }
}
```

Then register the job in the manifest file `AndroidManifest.xml` as follows:

```
<service
      android:exported="false"
      android:name=".MyJobService">
    <intent-filter>
        <action android:name=
            "com.firebase.jobdispatcher.ACTION_EXECUTE"
/>
    </intent-filter>
</service>
```

To include a check for availability, you have to perform the following steps:

1. Google Play Services needs to be added to the SDK installation. Inside Android Studio, go to Tools ➤ Android ➤ SDK Manager. In the menu choose Appearance & Behavior ➤ System Settings ➤ Android SDK. On the SDK Tools tab, select Google Play Services and then click the OK button.
2. Right-click the project, choose Open Module Settings, and in the menu select your app module. Go to the Dependencies tab, and add the library `com.google.android.gms:play-services` by clicking the + button.

To actually schedule a job from in your app, you can acquire the service, create a job, and then register this job by using this:

```
val gps = GoogleApiAvailability.getInstance().
      isGooglePlayServicesAvailable(this)
if(gps == ConnectionResult.SUCCESS) {
    // Create a new dispatcher using the Google Play
    // driver.
    val dispatcher = FirebaseJobDispatcher(
          GooglePlayDriver(this))

    val myJob = dispatcher.newJobBuilder()
            .setService(MyJobService::class.java)
               // the JobService that will be called
            .setTag("my-unique-tag")
               // uniquely identifies the job
            .build()

    dispatcher.mustSchedule(myJob)
} else {
    Log.e("LOG", "GooglePlayServices not available: " +
                GoogleApiAvailability.getInstance().
                getErrorString(gps))
}
```

This example scheduled a job with basic job scheduling characteristics. For more complex needs, the job builder allows for more options, as shown in Table 8-3. Just chain them before the `.build()` method.

Table 8-3. JobDispatcher Options

Method	Description
`setService(Class)`	This is the job class (in Kotlin you must write `MyService::class.java`).
`setTag(String)`	This uniquely identifies the job.
`setRecurring(Boolean)`	This sets whether this is a recurring job.
`setLifetime(Int)`	This sets the lifetime of the job. Possible values are `Lifetime.FOREVER` and `Lifetime.UNTIL_NEXT_BOOT`. With `FOREVER`, the job will persist even after a device reboot.
`setTrigger(Trigger)`	This sets when to trigger the job. Possible values are as follows: • `Trigger.NOW`: Starts the job immediately • `Trigger.executionWindow(windowStart: Int, windowEnd: Int)`: Sets an execution window (in seconds) • `Trigger.contentUriTrigger(uris: List<ObservedUri>)`: Watches content URIs
`setReplaceCurrent(Boolean)`	This specifies whether to replace an existing job, provided it has the same tag.
`setRetryStrategy(RetryStrategy)`	This sets the retry strategy. Possible values are as follows: • `RetryStrategy.DEFAULT_EXPONENTIAL`: Exponential, as in 30s, 1min, 2min, 4min, 8min, and so on. • `RetryStrategy.DEFAULT_LINEAR`: Linear, as in 30s, 60s, 90s, 120s, and so on.
`setConstraints(vararg Int)`	Here you can set constraints that need to be satisfied for the job to run. Possible values are as follows: • `Constraint.ON_ANY_NETWORK`: Run only if a network is available. • `Constraint.ON_UNMETERED_NETWORK`: Run only if an unmetered network is available. • `Constraint.DEVICE_CHARGING`: Run only if the device is plugged in. • `Constraint.DEVICE_IDLE`: Run only if the device is idle.
`setExtras(Bundle)`	Use this to set extra data. These will be passed to `onStartJob()` in the job service class.

Alarm Manager

If you need actions to happen at specific times, regardless of whether associated components are running, the Alarm Manager is the system service that you can use for such tasks.

As for matters concerning the Alarm Manager, your device is in one of the following states:

- **Device awake**

  The device is running. Usually this means also the screen is on, but there is no guarantee that if the screen is off, the device is no longer awake. Although often if the screen gets switched off, the device shortly after that leaves the awake state. The details depend on the hardware and the device's software configuration. The Alarm Manager can do its work if the device is awake, but being awake is not necessary for the Alarm Manager to fire events.

- **Device locked**

  The device is locked, and the user needs to unlock it before it can be handled again. A locked device *might* lead to the device going asleep; however, locking per se is a security measure and has no primary impact on the Alarm Manager's functioning.

- **Device asleep**

  The screen is switched off, and the device runs in a low-power consumption mode. Events triggered by the Alarm Manager will be able to wake up the device and then fire events, but this needs to be explicitly specified.

- **Device switched off**

  The Alarm Manager stops working and resumes working only the next time the device is switched on. Alarm events get lost when the device is switched off; there is nothing like a retry functionality here.

Alarm events are one of the following:

- A *PendingIntent* gets fired. Since PendingIntents may target at either services, activities, or broadcasts, an alarm event may start an activity or a service or send a broadcast.
- A *handler* gets invoked. This is a direct version of sending alarm events to the same component that is issuing the alarms.

To schedule alarms, you first need to get the Alarm Manager as a system service as follows:

```
val alrm = getSystemService(AlarmManager::class.java)
// or, if API level below 23:
// val alrm = getSystemService(Context.ALARM_SERVICE)
//       as AlarmManager
```

You can then issue alarms by various methods, as shown in Table 8-4. If for API levels 24 or higher you choose to have a listener receive alarm events, the details about how to use the associated handlers are covered in Chapter 10. If instead you're aiming at intents, all corresponding methods have a `type:Int` parameter with the following possible values:

- `AlarmManager.RTC_WAKEUP`

  The time parameter is wall clock time in UTC (milliseconds since January 1, 1970, 00:00:00); the device will be woken up if necessary.

- `AlarmManager.RTC`

  The time parameter is wall clock time in UTC (milliseconds since January 1, 1970, 00:00:00). If the device is asleep, the event will be discarded, and no alarm will be triggered.

- `AlarmManager.ELAPSED_REALTIME_WAKEUP`

  The time parameter is the time in milliseconds since the last boot, including sleep time. The device will be woken up if necessary

- `AlarmManager.ELAPSED_REALTIME`

  The time parameter is the time in milliseconds since the last boot, including the sleep time. If the device is asleep, the event will be discarded, and no alarm will be triggered.

***Table 8-4.** Issuing Alarms*

Method	Description
`set(type: Int, triggerAtMillis: Long, operation: PendingIntent): Unit`	This schedules an alarm. An intent gets invoked and triggered according to the type, and the time parameter is provided. Starting with API level 19, alarm event delivery might be inexact to optimize system resources usage. Use one of the `setExact` methods if you need exact delivery.
`set(type: Int, triggerAtMillis: Long, tag: String, listener: AlarmManager.OnAlarmListener, targetHandler: Handler): Unit`	This requires API level 24 or higher. It is a direct callback version of `set(Int, Long, PendingIntent)`. The `Handler` parameter can be `null` to invoke the listener on the app's main looper. Otherwise, the call of the listener will be performed from inside the handler provided.
`setAlarmClock(info: AlarmManager.AlarmClockInfo, operation: PendingIntent): Unit`	This requires API level 21 or higher. This schedules an alarm represented by an alarm clock. The alarm clock info object allows for adding an intent, which is able to describe the trigger. The system may choose to display relevant information about this alarm to the user. Other than that, this method is like `setExact(Int, Long, PendingIntent)` but implies the `RTC_WAKEUP` trigger type.

(*continued*)

Table 8-4. (*continued*)

Method	Description
`setAndAllowWhileIdle(type: Int, triggerAtMillis: Long, operation: PendingIntent): Unit`	This requires API level 23 or higher. Like `set(Int, Long, PendingIntent)`, but this alarm will be allowed to execute even when the system is in low-power idle modes.
`setExact(type: Int, triggerAtMillis: Long, operation: PendingIntent): Unit`	This requires API level 19 or higher. This schedules an alarm to be delivered precisely at the stated time.
`setExact(type: Int, triggerAtMillis: Long, tag: String, listener: AlarmManager.OnAlarmListener, targetHandler: Handler): Unit`	This requires API level 24 or higher. Direct callback version of `setExact(Int, Long, PendingIntent)`. The `Handler` parameter can be `null` to invoke the listener on the app's main looper. Otherwise, the call of the listener will be performed from inside the handler provided.
`setExactAndAllowWhileIdle( type: Int, triggerAtMillis: Long, operation: PendingIntent): Unit`	This requires API level 23 or higher. Like `setExact(Int, Long, PendingIntent)`, but this alarm will be allowed to execute even when the system is in low-power idle modes.
`setInexactRepeating(type: Int, triggerAtMillis: Long, intervalMillis: Long, operation: PendingIntent): Unit`	This schedules a repeating alarm that has inexact trigger time requirements; for example, an alarm that repeats every hour but not necessarily at the top of every hour.
`setRepeating(type: Int, triggerAtMillis: Long, intervalMillis: Long, operation: PendingIntent): Unit`	This schedules a repeating alarm. Starting at API level 19, this is the same as `setInexactRepeating()`.
`setWindow(type: Int, windowStartMillis: Long, windowLengthMillis: Long, operation: PendingIntent): Unit`	This schedules an alarm to be delivered within a given window of time.
`setWindow(int type: Int, windowStartMillis: Long, windowLengthMillis: Long, tag: String, listener: AlarmManager.OnAlarmListener, targetHandler: Handler) : Unit`	This requires API level 24 or higher. This is a direct callback version of `setWindow(int, long, long, PendingIntent)`. The `Handler` parameter can be `null` to invoke the listener on the app's main looper. Otherwise, the call of the listener will be performed from inside the handler provided.

The AlarmManager also has a couple of auxiliary methods, as described in Table 8-5.

Table 8-5. Auxiliary AlarmManager Methods

Method	Description
`cancel(operation: PendingIntent) : Unit`	This removes any alarms with a matching intent.
`cancel(listener: AlarmManager.OnAlarmListener): Unit`	This removes any alarm scheduled to be delivered to the given `AlarmManager.OnAlarmListener`.
`getNextAlarmClock() : AlarmManager.AlarmClockInfo`	This gets information about the next alarm clock currently scheduled.
`setTime(long millis): Unit`	This sets the system wall clock time, UTC (milliseconds since January 1, 1970, 00:00:00).
`setTimeZone(String timeZone): Unit`	This sets the system's persistent default time zone.

Loaders

Loaders are for loading data in the background. The main usage pattern is as follows:

1. The need to load data in a presumably time-consuming process arises, either in the UI thread after, for example, clicking a button or from any other place inside the code. Because the loading is expected to take some time, you want to have the loading happen in the background, not disturbing the UI, for example.
2. You get the `LoaderManager` from the context. From within activities, in Kotlin you just use the pseudo-getter `loaderManager`.
3. You implement and provide a subclass of `LoaderManager.LoaderCallbacks`. The main responsibility of this class consists of constructing an `android.content.Loader` and providing loading state callback functions.
4. You call `init(...)` on the `LoaderManager` and pass the callback's implementation.
5. You react on callback events.

Looking at the online API documentation for the Loader framework, two points are worth mentioning.

- Almost all over the description and for all examples (and also for almost all examples you can find on the Web), using the compatibility libraries for the Loader framework classes is suggested. This is for backward compatibility. The truth is, you don't have to do that. The Loader framework has been around for quite a while, since API level 11 to be precise, and since you might not care about the less than 1 percent in-use versions below API 11, the need to use the compatibility libraries for Loader framework classes might not be too high.

- Reading the documentation, it seems necessary to use loaders only in conjunction with fragments. The truth is, the Loader framework has nothing to do with fragments *per se*; you can use fragments, if you like, but you don't have to. So, you can use loaders with standard activities as well.

In the following paragraphs, we present a basic example for using the Loader framework. Experiment with it and extend it according to your needs.

As mentioned, inside an activity we have a `LoaderManager` already at hand; just use `loaderManager`, which by Kotlin gets internally transcribed to `getLoaderManager()`.

Next we provide an implementation of `LoaderManager.LoaderCallbacks`. You can use your own class, but for simplicity you can implement it directly over your activity as follows:

```
class MainActivity : AppCompatActivity(),
        LoaderManager.LoaderCallbacks<MyData> {
    val LOADER_ID = 42
    var loaded:MyData? = null

    // other fields and methods...

    override fun onCreateLoader(id: Int, args: Bundle?):
          Loader<MyData>? {
        Log.e("LOG", "onCreateLoader()")
        return makeLoader()
    }

    override fun onLoadFinished(loader: Loader<MyData>?,
          data: MyData?) {
        Log.e("LOG", "Load finished: " + data)
        loaded = data
        // show on UI or other actions...
    }

    override fun onLoaderReset(loader: Loader<MyData>?) {
        Log.e("LOG", "onLoaderReset()")
        loaded = null
        // remove from UI or other actions...
    }
}
```

What we have here is the following:

- The `LOADER_ID` is a unique ID for a loader. An app might have several loaders at work, so the Loader framework needs to be able to distinguish among different loaders.
- `var loaded:MyData? = null` will later receive the result from the loading process. Note that it is not necessary to keep a reference to the `Loader` itself, and you actually shouldn't do it, because the Loader framework will be taking care of the lifecycle of the loaders.

- The methods `onCreateLoader()`, `onLoadFinished()`, and `onLoadReset()` describe the implementation of `LoaderManager.LoaderCallbacks`. Note that the latter two are listeners, while the first one, with its name a little confusing, is a factory method for creating loaders. The framework will be taking care of `onCreateLoader()` invoked only when a loader needs to be constructed. If a loader with some ID exists and is not abandoned, it will be reused, and this method will not be called.

In our activity, we place the method `makeLoader()` to build a loader. `android.content.Loader` needs to be subclassed to have a usable loader. Two implementations are provided: `android.content.AsyncTaskLoader` and `android.content.CursorLoader`. The loader `CursorLoader` can be used to load table-like content from a content provider, and `AsyncTaskLoader` is more general and will be loading its data from inside an `AsyncTask`. We use the latter one for the example shown here:

```
fun makeLoader():Loader<MyData> {
  val res =
  @SuppressLint("StaticFieldLeak")
  object : AsyncTaskLoader<MyData>(this@MainActivity) {
      val myData: MutableList<String> =
            ArrayList<String>()
      var initLoaded = false

      override fun loadInBackground(): MyData {
           Log.e("LOG",
                "AsyncTaskLoader.loadInBackground()")
                Log.e("LOG", "Thread: " +
                      Thread.currentThread().toString())
                for (i in 0..9) {
                   Log.e("LOG", i.toString())
                    myData.add("Item " + i.toString())
                    Thread.sleep(1000)
                   if (isLoadInBackgroundCanceled)
                       throw OperationCanceledException(
                             "Canceled")
                }
                return MyData(myData)
      }

      override fun onStartLoading() {
           Log.e("LOG",
                "AsyncTaskLoader.onStartLoading()")
          super.onStartLoading()
           if (!initLoaded)
              forceLoad()
           initLoaded = true
      }
  }
  return res
}
```

Here are a few notes:

- `@SuppressLint("StaticFieldLeak")` will suppress the warning about possible memory leakage given inside Android Studio. The loader lifecycle is governed by the Loader framework, and `makeLoader()` will return a reusable loader, so the possible leak is mitigated. Instead, by moving it to a static field, which in Kotlin means providing it as an *object*, is not easy to do here since we need a reference to the activity for constructing the `AsycTaskLoader`.
- We provide for the Boolean `initLoaded` field to make sure the loading will be actually started by calling `forceLoad()` the first time.
- By design, the `loadInBackground()` method gets called in a background thread. This is where the loading actually happens. In the example we just count from 0 to 9. In a real-world scenario, you will of course do more interesting things here.
- To help the framework maintain a correct loader state, inside `loadInBackground()` you should regularly check `isLoadInBackgroundCanceled` and act accordingly. In the example, we throw an `OperationCanceledException`, which will not break your app but will be handled by the Loader framework. In fact, it will be transported up and eventually call the `onLoaderReset()` callback method.
- The method `onStartLoading()` gets called by the framework; you don't have to do that yourself.

All that is missing now is to start and maybe stop the loading. If you use two buttons for that in the UI, the corresponding methods read as follows:

```
fun go(view: View) {
    loaderManager.initLoader(LOADER_ID,null,this)
}

fun dismiss(view: View) {
    loaderManager.getLoader<MyData>(LOADER_ID)?.
         cancelLoad()
    loaderManager.destroyLoader(LOADER_ID)
}
```

The `cancelLoad()` method is necessary to tell the loader to cancel its loading operation, and the `destroyLoader()` method will unregister the loader from the Loader framework.

Notifications

A notification is a message an app can present to the user outside its normal GUI flow. Notifications show up in a special region of the screen, most prominently inside the status bar and notification drawer on top of a screen, in special dialogs, on the lock screen, on a paired Android Wear device, or on an app icon badge. See Figures 8-1 and 8-2 for a smartphone example. There you can see the notification icon and the notification content after the user expands the notification drawer.

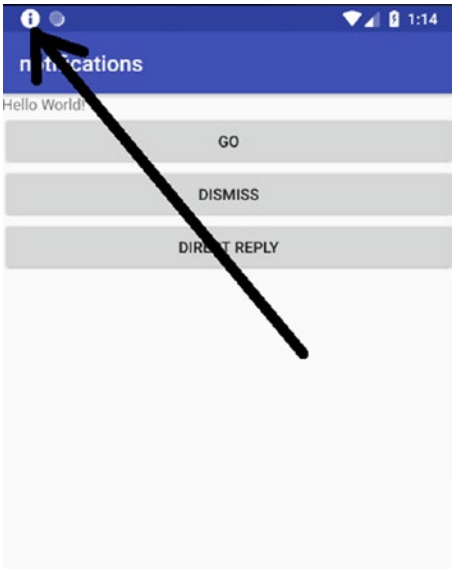

Figure 8-1. Smartphone notification

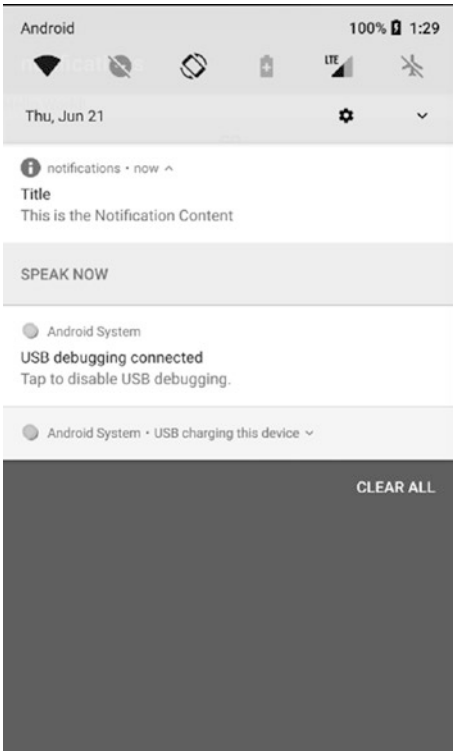

Figure 8-2. Notification content

Notifications also allow for actions, such as calling custom activities when tapped, or they can contain special action buttons or even edit fields a user can fill out. Likewise, although notifications were primarily built to show only short text snippets, with current Android versions, it is possible to present longer text there as well.

The online API documentation suggests to use the NotificationCompat API from the support library. Using this compatibility layer allows older versions to present similar or no-op variants on features that were made available only later, simplifying the development. Although using this compatibility layer removes the burden from the developer of presenting many branches inside the code to take care of different Android API levels, caution must be taken to not make an app unusable by depending too much on the latest notification API features.

To make sure the compatibility API is available for your project inside Android Studio, check the `build.gradle` setting of your module in the `dependencies` section (only one line; remove the newline after `implementation`).

```
implementation
    "com.android.support:support-compat:27.0.2"
```

The following sections present an outline of the notification API. With this API having grown considerably during the last years, please consult the online documentation for a more detailed description of all notification features.

Creating and Showing Notifications

To create and show a notification, you prepare action intents for a tap and additional action buttons, use a notification builder to construct the notification, register a notification channel, and finally make the framework show the notification. An example looks like this:

```
val NOTIFICATION_CHANNEL_ID = "1"
val NOTIFICATION_ID = 1

// Make sure this Activity exists
val intent = Intent(this, AlertDetails::class.java)
intent.flags = Intent.FLAG_ACTIVITY_NEW_TASK
      //or Intent.FLAG_ACTIVITY_CLEAR_TASK
val tapIntent = PendingIntent.getActivity(this, 0,
      intent, 0)

// Make sure this broadcast receiver    exists and can
// be called by explicit Intent like this
val actionIntent = Intent(this, MyReceiver::class.java)
actionIntent.action = "com.xyz.MAIN"
actionIntent.putExtra(EXTRA_NOTIFICATION_ID, 0)
val actionPendingIntent =
   PendingIntent.getBroadcast(this, 0, actionIntent, 0)

val builder = NotificationCompat.Builder(this,
      NOTIFICATION_CHANNEL_ID)
   .setSmallIcon( ... an icon resource id... )
```

```
   .setContentTitle("Title")
   .setContentText("Content Content Content Content ...")
   .setPriority(NotificationCompat.PRIORITY_DEFAULT)
   // add the default tap action
   .setContentIntent(tapIntent)
   .setAutoCancel(true)
   // add a custom action button
   .addAction( ... an icon resource id ...,
       "Go",
       actionPendingIntent)

buildChannel(NOTIFICATION_CHANNEL_ID)

val notificationManager =
      NotificationManagerCompat.from(this)
notificationManager.notify(
      NOTIFICATION_ID, builder.build())
```

The function `buildChannel()` is needed for Android API levels 26 and higher (Android 8.0). It reads as follows:

```
fun buildChannel(channelId:String) {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    // Create the NotificationChannel, but only
    // on API 26+ only after that it is needed
    val channel = if (Build.VERSION.SDK_INT >=
                Build.VERSION_CODES.O) {
        NotificationChannel(channelId,
              "Channel Name",
               NotificationManager.IMPORTANCE_DEFAULT)
    } else {
        throw RuntimeException("Internal error")
    }
    channel.description = "Description"
    // Register the channel with the system
    val notificationManager =
      if (Build.VERSION.SDK_INT >=
               Build.VERSION_CODES.M) {
        getSystemService(
              NotificationManager::class.java)
      } else {
        throw RuntimeException("Internal error")
      }
    notificationManager.
          createNotificationChannel(channel)
  }
}
```

An explanation for the other code follows:

- The notification itself needs a unique ID; we save that inside `NOTIFICATION_ID`.
- The action button, here for sending a broadcast, is for the example only. Having no action button is allowed.
- `setAutoCancel(true)` will lead to automatically dismissing the notification once the user taps the notification. This works only if `setContentIntent()` is used as well.
- Creating the notification channel is necessary only for API level 26 or higher (Android 8.0). The superfluous checks inside the outer `if` are necessary to make Android Studio not complain about compatibility issues.
- For all the strings you should use resource IDs where feasible; otherwise, use texts that better suit your needs.

Adding Direct Reply

Starting with API level 24 (Android 7.0), you can allow the user to enter text as a reply to a notification message. A major use case for this is of course a notification message from a messaging system like a chat client or e-mail. See Figure 8-3 for an example.

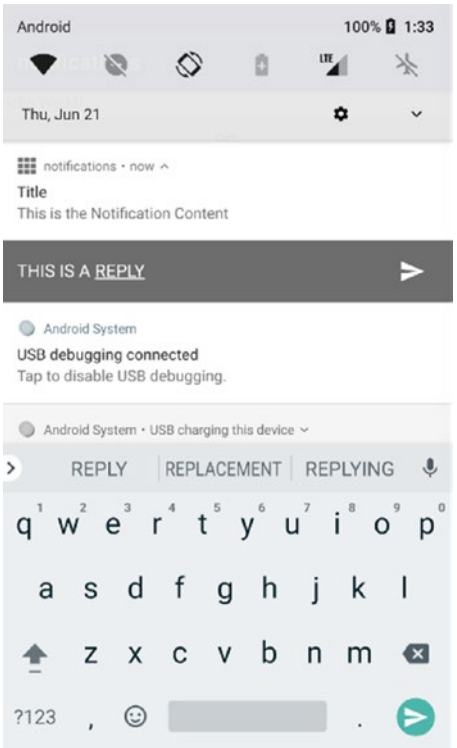

Figure 8-3. Reply notification

Since API levels below 24 are not able to provide that, your app should not rely on this functionality. Usually this is easy to achieve. For API levels 23 or lower, the activity called by tapping a notification can of course contain a facility to reply if needed.

A method that issues a notification with reply functionality might look like this:

```
fun directReply(view:View) {
    // Key for the string that's delivered in the
    // action's intent.
    val KEY_TEXT_REPLY = "key_text_reply"
    val remoteInput = RemoteInput.Builder(KEY_TEXT_REPLY)
            .setLabel("Reply label")
            .build()

    // Make sure this broadcast receiver exists
    val CONVERSATION_ID = 1
    val messageReplyIntent =
          Intent(this, MyReceiver2::class.java)
    messageReplyIntent.action = "com.xyz2.MAIN"
    messageReplyIntent.putExtra("conversationId",
         CONVERSATION_ID)

    // Build a PendingIntent for the reply
    // action to trigger.
    val replyPendingIntent = PendingIntent.
          getBroadcast(applicationContext,
              CONVERSATION_ID,
              messageReplyIntent,
               PendingIntent.FLAG_UPDATE_CURRENT)

    // Create the reply action and add the remote input.
    val action = NotificationCompat.Action.Builder(
          ... a resource id for an icon ...,
          "Reply", replyPendingIntent)
          .addRemoteInput(remoteInput)
          .build()

    val builder = NotificationCompat.Builder(this,
          NOTIFICATION_CHANNEL_ID)
    .setSmallIcon(... a resource id for an icon ...)
    .setContentTitle("Title")
    .setContentText("Content Content Content ...")
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)
    // add a reply action button
    .addAction(action)

    buildChannel(NOTIFICATION_CHANNEL_ID)

    val notificationManager =
          NotificationManagerCompat.from(this)
    notificationManager.notify(
           NOTIFICATION_ID, builder.build())
}
```

Here are a few notes about this code:

- KEY_TEXT_REPLY is used to identify the reply text in the intent receiver.
- CONVERSATION_ID is used to identify the conversation chain; here the notification and the intent that receives the reply must know they refer to each other.
- As usual, make sure that in production code you use string resources and appropriate text.

When the notification shows up, it will contain a Reply button, and when the user clicks it, the system will prompt for some reply text, which is then going to be sent to the intent receiver (messageReplyIntent in the example).

The intent receiver for the reply text might then have a receive callback that looks like this:

```
override fun onReceive(context: Context,
          intent: Intent) {
    Log.e("LOG", intent.toString())
    val KEY_TEXT_REPLY = "key_text_reply"

    val remoteInput = RemoteInput.
          getResultsFromIntent(intent)
    val txt = remoteInput?.
          getCharSequence(KEY_TEXT_REPLY)?:"undefined"
    val conversationId =
          intent.getIntExtra("conversationId",0)
    Log.e("LOG","reply text = " + txt)

    // Do s.th. with the reply...

    // Build a new notification, which informs the user
    // that the system handled their interaction with
    // the previous notification.
    val NOTIFICATION_CHANNEL_ID = "1"
    val repliedNotification =
          NotificationCompat.Builder(context,
                NOTIFICATION_CHANNEL_ID)
      .setSmallIcon(android.R.drawable.ic_media_play)
      .setContentText("Replied")
      .build()

    buildChannel(NOTIFICATION_CHANNEL_ID)

    // Issue the new notification.
    val notificationManager =
          NotificationManagerCompat.from(context)
    notificationManager.notify(conversationId,
          repliedNotification)
}
```

Here are some notes about this method:

- Fetches the reply text by using `RemoteInput.getResultsFromIntent()` using the same key as used for the reply input
- Fetches the conversation ID we added as an extra value to the intent
- Does whatever is appropriate to handle the reply
- Issues a reply to the reply by setting another notification

Notification Progress Bar

To add a progress bar to the notification, add the following to the builder with `PROGRESS_MAX` as the maximum integer value and `PROGRESS_CURRENT 0` at the beginning:

```
.setProgress(PROGRESS_MAX, PROGRESS_CURRENT, false)
```

Or, if you want an indeterminate progress bar, you instead use the following:

```
.setProgress(0, 0, true)
```

In a background thread that does the work, you then update a determinate progress bar by periodically executing the following with new `currentProgress` values:

```
builder.setProgress(PROGRESS_MAX, currentProgress, false)
notificationManager.notify(
      NOTIFICATION_ID, builder.build())
```

To finish a determinate or an indeterminate progress bar, you can write the following:

```
builder.setContentText("Download complete")
       .setProgress(0,0,false)
notificationManager.notify(
      NOTIFICATION_ID, builder.build())
```

Expandable Notifications

Notifications don't have to contain short messages only; using the expandable features, it is possible to show larger amounts of information to the user.

For details on how to do that, please consult the online documentation. Enter, for example, *android create expandable notification* in your favorite search engine to find the corresponding pages.

Rectifying Activity Navigation

To improve the user experience, an activity that was started from inside a notification can have its expected task behavior added. For example, if you click the back button, the activity down in the stack gets called. For this to work, you must define an activity hierarchy inside `AndroidManifest.xml`, for example, as follows:

```
<activity
    android:name=".MainActivity"
    ... >
</activity>
<!-- MainActivity is the parent for ResultActivity -->
<activity
    android:name=".ResultActivity"
    android:parentActivityName=".MainActivity" />
    ...
</activity>
```

You can then use a `TaskStackBuilder` to inflate a task stack for the intent called.

```
// Create an Intent for the Activity you want to start
val resultIntent =
      Intent(this, ResultActivity::class.java)
// Create the TaskStackBuilder
val stackBuilder = TaskStackBuilder.create(this)
stackBuilder.
      addNextIntentWithParentStack(resultIntent)
// Get the PendingIntent containing the back stack
val resultPendingIntent =
      stackBuilder.getPendingIntent(
           0, PendingIntent.FLAG_UPDATE_CURRENT)
// -> this can go to .setContentIntent() inside
//    the builder
```

For more details about activities and task management, please see Chapter 3.

Grouping Notifications

Beginning with API level 24 (Android 7.0), notifications can be grouped to improve the representation of several notifications that are related in some way. To create such a group, all you have to do is add the following to the builder chain, with `GROUP_KEY` being a string of your choice:

```
.setGroup(GROUP_KEY)
```

If you need a custom sorting, the default is to sort by incoming date; you can use the method `setSortKey()` from the builder. Sorting then happens lexicographically given that key. A grouping inside the notification drawer might look like Figure 8-4.

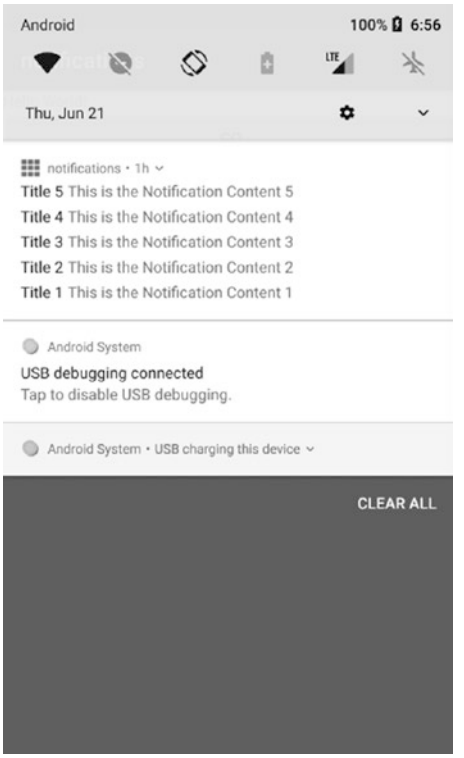

***Figure 8-4.** Notification group*

For API levels below 24 where some kind of Android-managed autosummary for groups is not available, you can add a notification summary. To do so, just create a notification like any other notification, but additionally call `.setGroupSummary(true)` inside the builder chain. Make sure all the notifications from the group *and* the summary use the same `setGroup(GROUP_KEY)`.

> **Caution** Because of a bug at least in API level 27 you *must* add a summary notification for the grouping to be enabled at all. So, the advice is, no matter what API level you are targeting, add a notification summary.

For the summary you might want to tailor the display style for displaying an appropriate number of summary items. For this aim, you can use a construct like the following inside the builder chain:

```
.setStyle(NotificationCompat.InboxStyle()
                .addLine("MasterOfTheUniverse Go play PacMan")
                .addLine("Silvia Cheng        Party tonite")
                .setBigContentTitle("2 new messages")
                .setSummaryText("xyz@example.com"))
```

Notification Channels

Starting with Android 8.0 (API level 26), another way of grouping notifications by *notification channel* has been introduced. The idea is to give the device user more control over how notifications get categorized and prioritized by the system, as well as the way notifications get presented to the user.

To create a notification channel, you write the following, which we have already seen in the preceding sections:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    // Create the NotificationChannel, but only
    // on API 26+ only after that it is needed
    val channel = if (Build.VERSION.SDK_INT >=
                Build.VERSION_CODES.O) {
        NotificationChannel(channelId,
              "Channel Name",
               NotificationManager.IMPORTANCE_DEFAULT)
    } else {
        throw RuntimeException("Internal error")
    }
    channel.description = "Description"
    // Register the channel with the system
    val notificationManager =
      if (Build.VERSION.SDK_INT >=
              Build.VERSION_CODES.M) {
        getSystemService(
              NotificationManager::class.java)
      } else {
        throw RuntimeException("Internal error")
      }
    notificationManager.
           createNotificationChannel(channel)
}
```

Speaking of Kotlin language styling, this looks a little clumsy. The superfluous `if` constructs were introduced so Android Studio wouldn't complain about compatibility issues. Adapt the channel ID, the channel name, and the importance in the channel constructor according to your needs, just as the description text.

By the way, the `createNotificationChannel()` method in the last line is idempotent. If a channel with the same characteristics already exists, nothing will happen.

The possible importance levels in the `NotificationChannel` constructor are `IMPORTANCE_HIGH` for sound and heads-up notification, `IMPORTANCE_DEFAULT` for sound, `IMPORTANCE_LOW` for no sound, and `IMPORTANCE_MIN` for neither sound nor status bar presence.

Having said that, it is up to the user how notification channels get handled. In your code you can still read the settings a user has made by using one of the `get*()` methods of the `NotificationChannel` object you can get from the manager via `getNotificationChannel()` or `getNotificationChannels()`. Please consult the online API documentation for details.

There is also a notification channel settings UI you can call by using this:

```
val intent = Intent(
      Settings.ACTION_CHANNEL_NOTIFICATION_SETTINGS)
intent.putExtra(Settings.EXTRA_APP_PACKAGE,
      getPackageName())
intent.putExtra(Settings.EXTRA_CHANNEL_ID,
      myNotificationChannel.getId())
startActivity(intent)
```

You can further organize notification channels by gathering them in groups, for example to separate work-related and private type channels. To create a group, you write the following:

```
val groupId = "my_group"
// The user-visible name of the group.
val groupName = "Group Name"
val notificationMngr =
    getSystemService(Context.NOTIFICATION_SERVICE)
    as NotificationManager
notificationMngr.createNotificationChannelGroup(
      NotificationChannelGroup(groupId, groupName))
```

You can then add the group to each notification channel by using its `setGroup()` method.

Notification Badges

Starting with Android 8.0 (API level 26), once a notification arrives in the system, a *notification badge* will show up in the app's icon. See, for example, Figure 8-5.

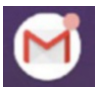

***Figure 8-5.** A notification badge*

You can control this badge using one of the `NotificationChannel` methods listed in Table 8-6.

***Table 8-6.** Notification Badges*

Method	Description
`setShowBadge(Boolean)`	This specifies whether to show the badge.
`setNumber(Int)`	Long-tapping an app icon with a badge will show the number of notifications that have arrived. You can tailor this number according to your needs by using this method.
`setBadgeIconType(Int)`	Long-tapping an app icon with a badge will show an icon associated with the notification. You can tailor the icon's size by using this method. Possible values are given as constants in class `NotificationCompat`: `BADGE_ICON_NONE`, `BADGE_ICON_SMALL`, and `BADGE_ICON_LARGE`.

Contacts

Managing and using contacts is one of the tasks a handheld device must really be good at. After all, handheld devices and especially smartphones get often used to communicating with other people, and contacts are abstracted entities representing people, groups, companies, or other "things" you use as address points for communication needs.

With contacts being so important, the built-in contacts framework has become quite complex over the history of Android. Fortunately, the complexity can be reduced somewhat if we restrict ourselves to looking solely at the back-end part and omit user interface peculiarities that are described in other chapters of this book. What is left for the description of the contacts framework is the following:

- Looking at the internals, especially the database model used
- Finding out how to read contacts data
- Finding out how to write contacts data
- Calling system activities to handle single contacts
- Synchronizing contacts
- Using quick contact badges

Contacts Framework Internals

The basic class to communicate with the contents framework is the `android.content.ContentResolver` class. This makes a lot of sense, since contact data fits well into what content providers deal with. You thus often use content provider operations to handle contact data. See Chapter 6 for more information.

The data model consists of three main tables: `Contacts`, `Raw Contacts`, `Data`. In addition, a couple of auxiliary tables for administrative tasks exist. You usually don't have to deal with any kind of direct table access, but in case you are interested, take a look at the online contacts framework documentation and the documentation for the `ContactsContract` class, which extensively describes the content provider contract for the contacts.

If you want to look at the contacts tables directly, using ADB for a virtual or rooted device, you can create a shell access to your device by using `cd SDK_INST/platform-tools ; ./adb root ; ./adb shell` in a terminal; see Chapter 18 for more information, and from there investigate the tables as follows:

```
cd /data
find . -name 'contacts*.db'
# <- this is to locate the contacts DB
cd <folder-for-contacts-db>
sqlite3 <name-of-contacts-db-file>
```

For example, enter `.header on` to switch on table header output, `.tables` to list all table names, and `select * from raw_contacts;` to list the `Raw Contacts` table.

Reading Contacts

For reading a number of contacts based on some criterion, you should create a loader as described in the "Loaders" section. To improve the code quality a little bit, we put the loading responsibility on our own class and write the following:

```
import android.app.Activity
import android.app.LoaderManager
import android.content.CursorLoader
import android.content.Loader
import android.database.Cursor
import android.os.Bundle
import android.provider.ContactsContract
import android.util.Log
import android.net.Uri.withAppendedPath

class ContactsLoader(val actv: Activity?,
                     val search:String):
         LoaderManager.LoaderCallbacks<Cursor> {
    override fun onCreateLoader(id: Int, args: Bundle?):
             Loader<Cursor>? {
        Log.e("LOG", "onCreateLoader()")

        val PROJECTION = arrayOf(
            ContactsContract.Contacts._ID,
            ContactsContract.Contacts.LOOKUP_KEY,
            ContactsContract.Contacts.DISPLAY_NAME_PRIMARY)

        val SELECTION =
            ContactsContract.Contacts.DISPLAY_NAME_PRIMARY
            + " LIKE ?"
        val selectionArgs = arrayOf("%" + search + "%")

        val contentUri =
             ContactsContract.Contacts.CONTENT_URI
        Log.e("LOG", contentUri.toString())

        // Starts the query
        return CursorLoader(
                 actv,
                 contentUri,
                 PROJECTION,
                 SELECTION,
                 selectionArgs,
                 null
        )
    }

    override fun onLoadFinished(loader: Loader<Cursor>,
                                data: Cursor) {
         Log.e("LOG", "Load finished: " + data)
         if(data.moveToFirst()) {
```

```
            do {
                Log.e("LOG", "Entry:")
                data.columnNames.forEachIndexed { i, s ->
                    Log.e("LOG", " -> " + s + " -> "
                          + data.getString(i))
                }
            } while (data.moveToNext())
        }
        // show on UI or other actions...
    }

    override fun onLoaderReset(loader: Loader<Cursor>?) {
         Log.e("LOG", "onLoaderReset()")
        // remove from UI or other actions...
    }
}
```

By virtue of `ContactsContract.Contacts.CONTENT_URI` that we use here as a URI, this will do a search in the `Contacts` table, returning basic contacts data.

To initialize and start the loader, all that is left to do, for example, in your activity is this:

```
val searchStr = "" // or whatever
val ldr = ContactsLoader(this, searchStr)
loaderManager.initLoader(0, null, ldr)
```

If instead you want to do a search inside the `Data` table, which contains phone numbers, e-mail addresses, and more, you write the following in `ContactsLoader.onCreateLoader()`:

```
...
val PROJECTION = arrayOf(
        ContactsContract.Data._ID,
        ContactsContract.Data.DISPLAY_NAME_PRIMARY,
        ContactsContract.CommonDataKinds.Email.ADDRESS)

val SELECTION =
  ContactsContract.CommonDataKinds.Email.ADDRESS
      + " LIKE ? " + "AND "
      + ContactsContract.Data.MIMETYPE + " = '"
      + ContactsContract.
            CommonDataKinds.Email.CONTENT_ITEM_TYPE
      + "'"

val selectionArgs = arrayOf("%" + search + "%")

val contentUri = ContactsContract.Data.CONTENT_URI
Log.e("LOG", contentUri.toString())
...
```

There are also special URIs you can use. For example, for finding contacts by e-mail address, you could use the content URI `ContactsContract.CommonDataKinds.Email.CONTENT_URI`.

As a third possibility, the URI given by `ContactsContract.Contacts.CONTENT_FILTER_URI` allows for adding search criteria inside the URI instead of specifying them in the `CursorLoader` constructor.

```
...
val PROJECTION : Array<String>? = null
val SELECTION : String? = null
val selectionArgs : Array<String>? = null

val contentUri = Uri.withAppendedPath(
        ContactsContract.Contacts.CONTENT_FILTER_URI,
         Uri.encode(search))
Log.e("LOG", contentUri.toString())
...
```

Note that in this case it is not allowed to pass an empty string ("") as a search criterion.

Writing Contacts

Inserting or updating contacts best happens in batch mode. You start with a list of the item type `ContentProviderOperation` and fill it with operations as follows:

```
import android.content.Context
import android.content.ContentProviderOperation
import android.content.ContentResolver
import android.provider.ContactsContract
import android.content.ContentValues.TAG
import android.util.Log
import android.widget.Toast

class ContactsWriter(val ctx:Context, val contentResolver:
        ContentResolver) {
  val opList = mutableListOf<ContentProviderOperation>()

  fun addContact(accountType:String, accountName:String,
                 firstName:String, lastName:String,
                 emailAddr:String, phone:String) {
      val firstOperationIndex = opList.size
```

Inside this method we first create a new contact. The `Contacts` table will be filled automatically; direct access is not possible anyway. The device's user account and account type are needed; otherwise, the operations silently will fail!

```
// Creates a new raw contact.
var op = ContentProviderOperation.newInsert(
      ContactsContract.RawContacts.CONTENT_URI)
     .withValue(
        ContactsContract.RawContacts.ACCOUNT_TYPE,
        accountType)
     .withValue(
```

```
       ContactsContract.RawContacts.ACCOUNT_NAME,
        accountName)
opList.add(op.build())
```

Next, still inside the method, we create a display name for the new row. This is a row inside the table `StructuredName`.

```
// Creates the display name for the new row
    op = ContentProviderOperation.newInsert(
          ContactsContract.Data.CONTENT_URI)
       // withValueBackReference will make sure the
       // foreign key relations will be set
       // correctly
       .withValueBackReference(
          ContactsContract.Data.RAW_CONTACT_ID,
          firstOperationIndex)
       // The data row's MIME type is StructuredName
       .withValue(ContactsContract.Data.MIMETYPE,
         ContactsContract.CommonDataKinds.
              StructuredName.CONTENT_ITEM_TYPE)
        // The row's display name is the name in the UI.
        .withValue(ContactsContract.CommonDataKinds.
               StructuredName.DISPLAY_NAME,
               firstName + " " + lastName)
    opList.add(op.build())
```

Likewise, we add the phone number and the e-mail address.

```
// The specified phone number
op = ContentProviderOperation.newInsert(
      ContactsContract.Data.CONTENT_URI)
    // Fix foreign key relation
    .withValueBackReference(
      ContactsContract.Data.RAW_CONTACT_ID,
           firstOperationIndex)
    // Sets the data row's MIME type to Phone
    .withValue(ContactsContract.Data.MIMETYPE,
               ContactsContract.CommonDataKinds.
                     Phone.CONTENT_ITEM_TYPE)
    // Phone number and type
    .withValue(ContactsContract.CommonDataKinds.
          Phone.NUMBER, phone)
    .withValue(ContactsContract.CommonDataKinds.
          Phone.TYPE,
              android.provider.ContactsContract.
              CommonDataKinds.Phone.TYPE_HOME)
opList.add(op.build())

// Inserts the email
op = ContentProviderOperation.newInsert(
      ContactsContract.Data.CONTENT_URI)
// Fix the foreign key relation
.withValueBackReference(
```

```
      ContactsContract.Data.RAW_CONTACT_ID,
  firstOperationIndex)
// Sets the data row's MIME type to Email
.withValue(ContactsContract.Data.MIMETYPE,
            ContactsContract.CommonDataKinds.
               Email.CONTENT_ITEM_TYPE)
// Email address and type
.withValue(ContactsContract.CommonDataKinds.
      Email.ADDRESS, emailAddr)
.withValue(ContactsContract.CommonDataKinds.
      Email.TYPE,
  android.provider.ContactsContract.
      CommonDataKinds.Email.TYPE_HOME)
```

Finally, before closing the method, we add a yield point. This has no functional influence but introduces a break so the system can do other work to improve usability. The following snippet also contains the rest of the class:

```
    // Add a yield point.
    op.withYieldAllowed(true)

    opList.add(op.build())
}

fun reset() {
    opList.clear()
}

fun doAll() {
    try {
         contentResolver.applyBatch(
             ContactsContract.AUTHORITY,
             opList as ArrayList)
    } catch (e: Exception) {
        // Display a warning
        val duration = Toast.LENGTH_SHORT
        val toast = Toast.makeText(ctx,
            "Something went wrong", duration)
         toast.show()
        // Log exception
        Log.e("LOG", "Exception encountered "+
             "while inserting contact: " + e, e)
        }
    }
}
```

This uses a fixed phone type and a fixed e-mail type, but I guess you get the point. Also, make sure in productive code you use resource strings instead of hard-coded strings, as shown here. To use the class, all you have to do from inside an activity is the following:

```
val cwr = ContactsWriter(this, contentResolver)
cwr.addContact("com.google","user@gmail.com",
    "Peter","Kappa",
```

```
    "post@kappa.com","0123456789")
cwr.addContact("com.google","user@gmail.com",
    "Hilda","Kappa",
    "post2@kappa.com","0123456789")
cwr.doAll()
```

To update a contacts entry, we introduce another function inside the `ContactsWriter` class.

```
fun updateContact(id:String, firstName:String?,
    lastName:String?, emailAddr:String?, phone:String?) {
  var op : ContentProviderOperation.Builder? = null
  if(firstName != null && lastName != null) {
    op = ContentProviderOperation.newUpdate(
           ContactsContract.Data.CONTENT_URI)
      .withSelection(ContactsContract.Data.CONTACT_ID +
          " = ? AND " + ContactsContract.Data.MIMETYPE +
          " = ?",
          arrayOf(id, ContactsContract.CommonDataKinds.
                     StructuredName.CONTENT_ITEM_TYPE))
      .withValue(ContactsContract.Contacts.DISPLAY_NAME,
          firstName + " " + lastName)
    opList.add(op.build())
  }
  if(emailAddr != null) {
    op = ContentProviderOperation.newUpdate(
           ContactsContract.Data.CONTENT_URI)
      .withSelection(ContactsContract.Data.CONTACT_ID +
          " = ? AND " + ContactsContract.Data.MIMETYPE +
          " = ?",
           arrayOf(id, ContactsContract.CommonDataKinds.
                      Email.CONTENT_ITEM_TYPE))
      .withValue(ContactsContract.CommonDataKinds.Email.
                 ADDRESS, emailAddr)
    opList.add(op.build())
  }
  if(phone != null) {
    op = ContentProviderOperation.newUpdate(
           ContactsContract.Data.CONTENT_URI)
      .withSelection(ContactsContract.Data.CONTACT_ID +
          " = ? AND " + ContactsContract.Data.MIMETYPE +
          " = ?",
          arrayOf(id, ContactsContract.CommonDataKinds.
                     Phone.CONTENT_ITEM_TYPE))
      .withValue(ContactsContract.CommonDataKinds.Phone.
                 NUMBER, phone)
    opList.add(op.build())
  }
}
```

As an input, you need the ID key from the `Raw Contacts` table; any function argument not `null` gets updated. For example, you can write the following inside the activity:

```
val rawId = ...
val cwr = ContactsWriter(this, contentResolver)
cwr.updateContact(rawId, null, null,
                  "postXXX@kappa.com", null)
cwr.doAll()
```

As a last function, we add the possibility to delete a contact, again based on the ID.

```
fun delete(id:String) {
    var op = ContentProviderOperation.newDelete(
        ContactsContract.RawContacts.CONTENT_URI)
    .withSelection(ContactsContract.RawContacts.
          CONTACT_ID + " = ?",
          arrayOf(id))
    opList.add(op.build())
}
```

Using it is similar to an update.

```
val rawId = ...
val cwr = ContactsWriter(this, contentResolver)
cwr.delete(rawId)
cwr.doAll()
```

Using Contacts System Activities

To read or update a single contact, you can avoid having to write your own user interface. Just use the system activity to access a contact. An appropriate intent call for creating a single contact looks like this:

```
val intent = Intent(Intents.Insert.ACTION)
intent.setType(ContactsContract.RawContacts.CONTENT_TYPE)
intent.putExtra(Intents.Insert.EMAIL, emailAddress)
      .putExtra(Intents.Insert.EMAIL_TYPE,
 CommonDataKinds.Email.TYPE_WORK)
      .putExtra(Intents.Insert.PHONE, phoneNumber)
      .PutExtra(Intents.Insert.PHONE_TYPE, Phone.
 TYPE_WORK)
startActivity(intent)
```

This will open the contacts screen for creating a new contact and prefill the given fields.

To instead edit an existing contact, once you have the lookup key and the raw contact ID, as shown earlier, write the following:

```
val uri = Contacts.getLookupUri(id, lookupKey)
val intent = Intent(Intent.ACTION_EDIT)
// the following must be done in _one_ call, do not
// chain .setData() and .setType(), because they
// overwrite each other!
```

```
intent.setDataAndType(uri, Contacts.CONTENT_ITEM_TYPE)
// starting at API level 15, this is needed:
intent.putExtra("finishActivityOnSaveCompleted", true)

// now put any data to update, for example
intent.putExtra(Intents.Insert.EMAIL, newEmail)
...
startActivity(intent)
```

Synchronizing Contacts

At this place, we provide a brief outline on what to do if you want to write a contacts synchronization app between your device and a server.

1. Build a subclass of `android.app.Application` and register it as `name` inside the `<application>` tag of the file `AndroidManifest.xml`. Inside its `onCreate()` callback, instantiate a `SyncAdapter` and provide for a method a `SyncAdapter` service can fetch this instance.
2. Build a bindable service component the system can use for synchronization.
3. Implement the `SyncAdapter`, for example by subclassing an `AbstractThreadedSyncAdapter`.
4. Provide an XML file to tell the system about the adapter. The procedure gets described in the online API documentation of `AbstractThreadedSyncAdapter`.
5. Optionally provide a service for authentication. The `AccountManager` starts this service to begin the authentication process. When the system wants to authenticate a user account for the application's sync adapter, it calls the service's `onBind()` method to get an `IBinder` for the authenticator.
6. Optionally provide a subclass of `AbstractAccountAuthenticator`, which handles requests for authentication.

Using Quick Contact Badges

Quick contact badges allow you to use a GUI widget that your user can tap to see a contact's details and take any suitable action from there such as sending an e-mail, issuing a call, or whatever makes sense. This details screen gets presented by the system; you don't have to implement it in your app. See Figure 8-6.

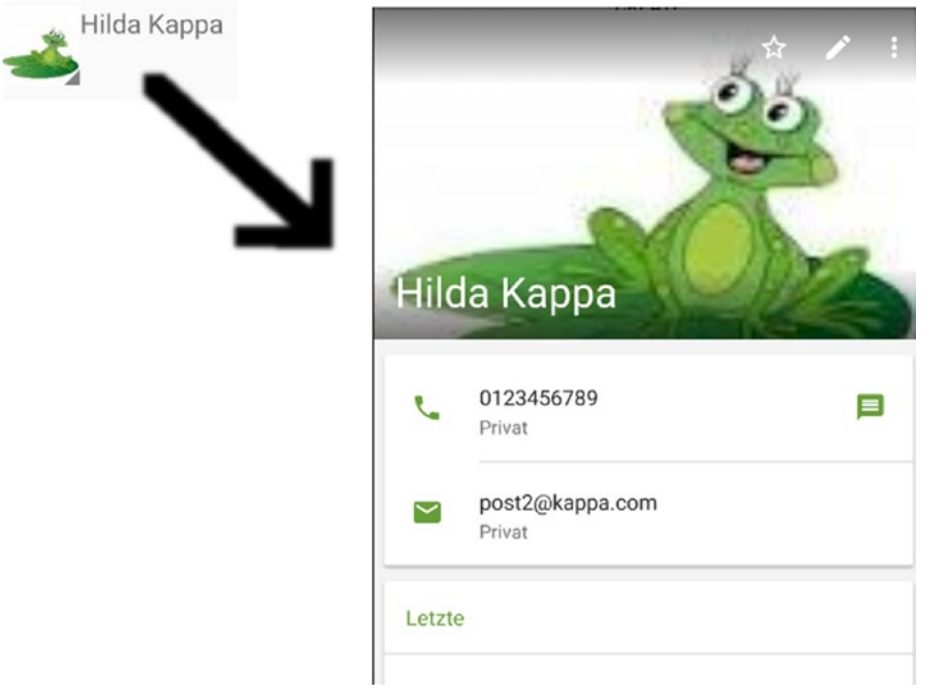

Figure 8-6. A quick contact badge

To generate such a quick contact badge, inside your layout file you must add the following:

```
<QuickContactBadge
          android:id="@+id/quickBadge"
          android:layout_width="60dp"
          android:layout_height="60dp"
          android:scaleType="centerCrop"/>
```

In you code you must connect the badge to the following information you get from the contacts provider: the raw contact ID, the lookup key, and a thumbnail URI. The corresponding code might look like this:

```
val id = row[ContactsContract.Contacts._ID]
val lookup = row[ContactsContract.Contacts.
      LOOKUP_KEY]
val photo = row[ContactsContract.Contacts.
      PHOTO_THUMBNAIL_URI]
```

Here, the `row`, for example, is a map you get from a contacts content provider query. In this case, a query in `Raw Contact` is enough; you don't need to also query the `Data` table.

From here we configure the badge as follows, for example after you load the contact information by user interface activities:

```
val contactUri = ContactsContract.Contacts.getLookupUri(
      id?.toLong()?:0,
      lookup)
quickBadge.assignContactUri(contactUri)
val thumbnail =
       loadContactPhotoThumbnail(photo.toString())
quickBadge.setImageBitmap(thumbnail)
```

Here, the `loadContactPhotoThumbnail()` function loads the thumbnail image data.

```
private fun loadContactPhotoThumbnail(photoData: String):
       Bitmap? {
  var afd: AssetFileDescriptor? = null
  try {
      val thumbUri = Uri.parse(photoData)
       afd = contentResolver.
            openAssetFileDescriptor(thumbUri, "r")
       afd?.apply {
           fileDescriptor?.apply {
               return BitmapFactory.decodeFileDescriptor(
                      this, null, null)
           }
       }
  } catch (e: FileNotFoundException) {
      // Handle file not found errors ...
  } finally {
      afd?.close()
  }
  return null
}
```

Search Framework

The search framework allows you to seamlessly add search functionality to your app and register your app as a searchable items provider in the Android OS.

Talking of the user interface, you have two options.

- Opening a search dialog
- Adding a search widget to your UI via `SearchView`

More precisely, to include search facilities inside your app, you have to do this:

1. Provide a searchable configuration as an XML file.
2. Provide an activity that (a) is able to receive a search query, (b) performs the search inside your app's data, and (c) displays the search result.
3. Provide a dialog or search widget.

The rest of this section walks through these requirements.

The Searchable Configuration

The searchable configuration is a file named `searchable.xml` that resides inside the folder `/res/xml` of your project. The most basic contents of this file read as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android=
      "http://schemas.android.com/apk/res/android"
  android:label="@string/app_label"
  android:hint="@string/search_hint" >
</searchable>
```

`@string/...` points to localized string resources.`@string/app_label` points to a label and should be equal to the name of the `label` attribute of the `<application>` element. The other one, `@string/search_hint`, is the string to be shown inside search fields if nothing has been entered yet. It is recommended and should show something like `Search <content>`, with `<content>` being specific to the data your app provides. There are a lot more possible attributes and some optional child elements; we will mention some in the following sections. For the complete list, please see the online documentation in the "Searchable Configuration" section.

The Searchable Activity

For the activity that handles search-related issues inside your app, start with its declaration inside `AndroidManifest.xml`. The activity needs to have a special signature there, as follows:

```
<activity android:name=".SearchableActivity" >
  <intent-filter>
    <action android:name=
          "android.intent.action.SEARCH" />
  </intent-filter>
  <meta-data android:name="android.app.searchable"
             android:resource="@xml/searchable"/>
</activity>
```

The name of the activity is up to you; all the other tags and attributes must be as shown here.

Next we let this activity receive the search request. This is done inside its `onCreate()` callback as follows:

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

  // This is the standard way a system search dialog
  // or the search widget communicates the query
  // string:
  if (Intent.ACTION_SEARCH == intent.action) {
    val query =
         intent.getStringExtra(SearchManager.QUERY)
    doMySearch(query)
  }

  // More initialization if necessary...
}
```

The `doMySearch()` function is supposed to perform the search and present the result inside the `SearchableActivity`. The way this is done is totally up to the application; it could be a database search or a search using a content provider or anything you want.

The Search Dialog

For any activity to open the system's search dialog and have it pass the query entered there to the `SearchableActivity`, you write the following in `AndroidManifest.xml`:

```
<activity android:name=".SearchableActivity" >
  <!-- same as above -->
</activity>
<activity android:name=".MainActivity"
          android:label="Main">
  <!-- ... -->

  <!-- Enable the search dialog and let it send -->
  <!-- the queries to SearchableActivity -->
  <meta-data android:name=
        "android.app.default_searchable"
             android:value=
        ".SearchableActivity" />
</activity>
```

This example allows for `MainActivity` to open the system's search dialog. In fact, you can use any suitable activity from inside your app for that purpose.

To open the search dialog inside your searching activity, write the following:

```
onSearchRequested()
```

> **Note** Usually directly executing obvious callback function starting with `on...` has a bad smell. You usually do that for half-legal shortcuts. The reason why we have to do it here is that your device might have a dedicated search button. In this case, `onSearchRequested()` gets called from the system, and it is a real callback method. Because such a button is optional, it is, however, necessary to always provide a search initiator from inside your app.

Figure 8-7 shows a dialog-based search flow.

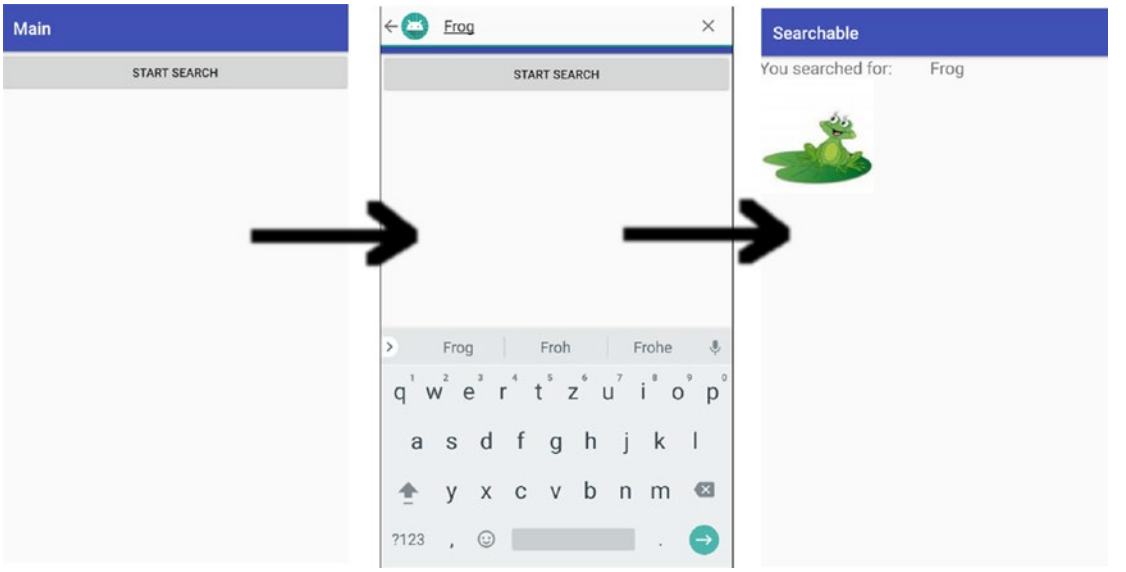

Figure 8-7. A dialog-based search flow

The Search Widget

Instead of opening the system search dialog, you can place a `<SearchView>` widget inside your UI. While in principle you can place it wherever you like, it is recommended you put it in the action bar. For this aim, provided you have set up an action bar and defined a menu there, inside the menu XKML definition you write the following:

```
<menu xmlns:android=
    "http://schemas.android.com/apk/res/android"
      xmlns:app=
    "http://schemas.android.com/apk/res-auto">

  <!-- Usually you have Settings in any menu -->
  <item android:id="@+id/action_settings"
        android:title="Settings"
         app:showAsAction="never"/>
```

```
  <item android:id="@+id/action_search"
        android:title="Search"
        app:showAsAction="ifRoom|collapseActionView"
        app:actionViewClass=
              "android.support.v7.widget.SearchView"
      android:icon=
              "@android:drawable/ic_menu_search"/>

  <!-- more items ... ->
</menu>
```

What then needs to be done inside your app to connect the widget with the search framework is the following:

```
// Set the searchable configuration
val searchManager = getSystemService(SEARCH_SERVICE)
      as SearchManager
val searchView = menu.findItem(R.id.action_search).
      actionView as SearchView
searchView.setSearchableInfo(
      searchManager.getSearchableInfo(componentName))
// Do not iconify the widget; expand it by default:
searchView.setIconifiedByDefault(false)
```

That is it! The flow looks like Figure 8-8 (you'd click the search icon in the action bar).

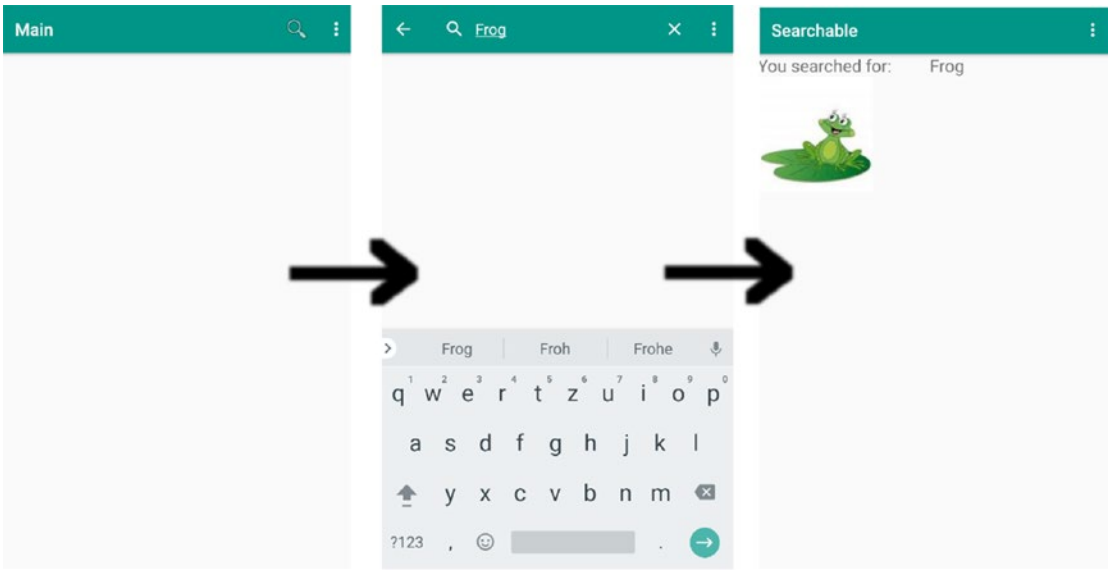

Figure 8-8. A widget-based search flow

Search Suggestions

There are two ways you can help the user input search query strings. You can let the system memorize queries for the next time the search gets used, and you can let your app provide fully customizable suggestions.

Recent Queries Suggestions

For the recent queries suggestions, you implement a content provider subclass of `SearchRecentSuggestionsProvider` and add it to `AndroidManifest.xml` like any other content provider. A basic but nevertheless already fully implemented content provider looks like this:

```
class RecentsProvider :
        SearchRecentSuggestionsProvider {
    val AUTHORITY = "com.example.RecentsProvider"
    val MODE = DATABASE_MODE_QUERIES

    init {
        setupSuggestions(AUTHORITY, MODE)
    }
}
```

Register it inside `/res/xml/searchable.xml` as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android=
        "http://schemas.android.com/apk/res/android"
     android:label="@string/app_label"
     android:hint="@string/search_hint"
     android:searchSuggestAuthority=
         "com.example.RecentsProvider"
     android:searchSuggestSelection=
         " ?">
</searchable>
```

New are the last two attributes. The `android:searchSuggestAuthority` attribute here draws a connection to the provider.

The content provider must still be registered in `AndroidManifest.xml`. This for example reads as follows:

```
<provider
    android:name=".RecentsProvider"
    android:authorities="com.example.RecentsProvider"
    android:enabled="true"
    android:exported="true">
</provider>
```

This reads previous queries from an automatically generated database. What is left to do is add the search queries. For this aim, inside the `SearchableActivity` class write the following:

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

  // This is the standard way a system search dialog
  // or the search widget communicates the query
  // string:
  if (Intent.ACTION_SEARCH == intent.action) {
    val query =
        intent.getStringExtra(SearchManager.QUERY)

    // Add it to the recents suggestion database
    val suggestions = SearchRecentSuggestions(this,
         RecentsProvider.AUTHORITY, RecentsProvider.MODE)
    suggestions.saveRecentQuery(q, null)

    doMySearch(query)
  }

  // More initialization if necessary...
}
```

The second parameter for the `saveRecentQuery()` method could be a second line for annotation purposes. For this to work, you have to use `val MODE = DATABASE_MODE_QUERIES` or `DATABASE_MODE_2LINES` in the `RecentsProvider` and find a way to retrieve annotation text inside the `SearchableActivity` class.

Custom Suggestions

Custom suggestions are more powerful compared to recents suggestions. They can be fully app or domain specific, and you can provide intelligent suggestions to the user, tailored for the current user action. Compared to recents suggestions, you instead implement and register a `ContentProvider` by obeying certain rules.

- The Android OS will fire `ContentProvider.query(uri, projection, selection, selectionArgs, sortOrder` calls with URIs like the following:

  ```
  content://your.authority/
      optional.suggest.path/
      SUGGEST_URI_PATH_QUERY/
      <query>
  ```

  Here, `your.authority` is the content provider authority, `/optional.suggest.path` might be added by the search configuration for disambiguation, and `SUGGEST_URI_PATH_QUERY` is the value of the constant `SearchManager.SUGGEST_URI_PATH_QUERY`. The `<query>` contains the string to be searched for. Both the `selection` and `selectionArgs` parameters will be filled only if appropriately configured in the search configuration.

- The resulting `Cursor` must return the following fields (shown are constant names):
  - `BaseColumns._ID`

    This is a (technically) unique ID you must provide.
  - `SearchManager.SUGGEST_COLUMN_TEXT_1`

    This is the search suggestion.
  - `SearchManager.SUGGEST_COLUMN_TEXT_2`

    (optional) This is a second, less important string representing an annotation text.
  - `SearchManager.SUGGEST_COLUMN_ICON_1`

    (optional) This is a drawable resource ID, content, or file URI string for an icon to be shown on the left.
  - `SearchManager.SUGGEST_COLUMN_ICON_2`

    (optional) This is a drawable resource ID, content, or file URI string for an icon to be shown on the right.
  - `SearchManager.SUGGEST_COLUMN_INTENT_ACTION`

    (optional) This is an intent action string that is used to call an intent when the suggestion gets clicked.
  - `SearchManager.SUGGEST_COLUMN_INTENT_DATA`

    This is an intent data member to be sent with the intent.
  - `SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID`

    This is a string to be appended to the intent `data` member.
  - `SearchManager.SUGGEST_COLUMN_INTENT_EXTRA_DATA`

    This is extra data to be sent with the intent.
  - `SearchManager.SUGGEST_COLUMN_QUERY`

    This is the original query string.
  - `SearchManager.SUGGEST_COLUMN_SHORTCUT_ID`

    This is used when providing suggestions for the *quick search box*. It indicates whether a search suggestion should be stored as a shortcut and whether it should be validated.
  - `SearchManager.SUGGEST_COLUMN_SPINNER_WHILE_ REFRESHING`

    This is used when providing suggestions for the quick search box; a spinner should be shown instead of the icon from `SUGGEST_COLUMN_ICON_2` while the shortcut of this suggestion is being refreshed in the quick search box.

Let's try to build a valid example for custom suggestions. We start with a working example of a recents suggestion provider as described earlier. It doesn't matter whether you use the dialog or the widget method. Note the differences are the content provider and the search configuration.

For a search configuration defined by file `/res/xml/searchable.xml`, enter the following:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android=
      "http://schemas.android.com/apk/res/android"
    android:label=
      "@string/app_label"
    android:hint=
      "@string/search_hint"
    android:searchSuggestAuthority=
      "com.example.CustomProvider"
    android:searchSuggestIntentAction=
      "android.intent.action.VIEW">
</searchable>
```

Next we define a new content provider.

```
class CustomProvider : ContentProvider() {
  override fun query(uri: Uri,
           projection: Array<String>?,
           selection: String?,
           selectionArgs: Array<String>?,
          sortOrder: String?): Cursor? {
     Log.e("LOG", "query(): " + uri +
             " - projection=" +
                Arrays.toString(projection) +
             " - selection=" + selection +
             " - selectionArgs=" +
                Arrays.toString(selectionArgs) +
             " - sortOrder=" + sortOrder)
     return null
  }

  override fun delete(uri: Uri, selection: String?,
        selectionArgs: Array<String>?): Int {
      throw UnsupportedOperationException(
          "Not yet implemented")
  }

  override fun getType(uri: Uri): String? {
      throw UnsupportedOperationException(
          "Not yet implemented")
  }

  override fun insert(uri: Uri, values: ContentValues?):
        Uri? {
      throw UnsupportedOperationException(
          "Not yet implemented")
  }
```

```
  override fun onCreate(): Boolean {
      return false
  }

  override fun update(uri: Uri, values: ContentValues?,
        selection: String?,
        selectionArgs: Array<String>?): Int {
      throw UnsupportedOperationException(
          "Not yet implemented")
  }
}
```

Register it in `AndroidManifest.xml`.

```
<provider
    android:name=".CustomProvider"
    android:authorities="com.example.CustomProvider"
    android:enabled="true"
    android:exported="true">
</provider>
```

This is what happens so far when the user starts a search:

1. Whenever the user enters or removes a character, the system will go to the search configuration it sees by looking at the `searchSuggestAuthority` attribute that it needs so it can address a content provider with this authority assigned to it.
2. By looking in `AndroidManifest.xml`, it sees that this authority is connected to the provider class `CustomProvider`.
3. It invokes a `query()` on this provider and expects a `Cursor` to return the custom suggestions.
4. If the user taps a suggestion, by virtue of the `searchSuggestIntentAction` attribute set to `android.intent.action.VIEW`, the `SearchableActivity`'s `onCreate()` will see the incoming intent with the `VIEW` action.

Up to now we let the `query()` method return `null`, which is equivalent to *no* suggestions, but we added a logging statement, so we can see what arrives at the `query()` method. For example, when the user enters `sp`, the arguments so far will read as follows:

```
query(): content://com.example.CustomProvider/
        search_suggest_query/sp?limit=50
  projection=null
  selection=null
  selectionArgs=null
  sortOrder=null
```

The arguments sent to the `query()` method by the search framework can be tailored extensively by various search configuration attributes. Now, however, we refer you to the online documentation and continue with extracting the information from the first `uri` parameter. How to build `Cursor` objects is described in Chapter 6; for this example, we use a `MatrixCursor`, and instead of returning `null`, we could, for example, return the following from inside the `SearchableActivity`:

```
override fun query(uri: Uri,
                   projection: Array<String>?,
                   selection: String?,
                   selectionArgs: Array<String>?,
                   sortOrder: String?): Cursor? {
      Log.e("LOG", "query(): " + uri +
              " - projection=" +
                Arrays.toString(projection) +
              " - selection=" + selection +
              " - selectionArgs=" +
                   Arrays.toString(selectionArgs) +
              " - sortOrder=" + sortOrder)

      val lps = uri.lastPathSegment // the query
      val qr = uri.encodedQuery     // e.g. "limit=50"

      val curs = MatrixCursor(arrayOf(
              BaseColumns._ID,
              SearchManager.SUGGEST_COLUMN_TEXT_1,
              SearchManager.SUGGEST_COLUMN_INTENT_DATA
      ))
      curs.addRow(arrayOf(1, lps + "-Suggestion 1",
           lps + "-Suggestion 1"))
      curs.addRow(arrayOf(2, lps + "-Suggestion 2",
           lps + "-Suggestion 2"))

      return curs
}
```

This example provides only silly suggestions; you can write something more clever to the `MatrixCursor`.

As a last modification, you can make your search suggestions available to the system's *quick search box*. All you have to do for that is add `android:includeInGlobalSearch = true` to your search configuration. The user must allow this inside the settings for this connection to take effect.

Location and Maps

Handheld devices may track their geographic position, and they may interact with map services to graphically interfere with a user's location needs. Geographical position is not only about latitude and longitude numbers but also about finding out street addresses and points of interest. While it is certainly an important ethical question how far apps can go to track their users' personal life, the possibilities for interesting apps and games are potentially endless. In this section, we talk about the technical possibilities. Just be cautious with your user's data and be transparent with what you are doing with them.

The Android OS itself contains a location framework with classes in the package `android.location`. However, the official position of Google is to favor the Google Play Services Location API because it is more elaborate and simpler to use. We follow this suggestion and talk about the services API in the following paragraphs. Location is about finding out the geographical position of a device as a latitude-longitude pair and finding out the street names, house numbers, and other points of interest given the geographical position.

To make the Services Location API available to your app, add the following as dependencies in your app module's `build.gradle` file (as just two lines; remove the newlines after `implementation`):

```
implementation
   'com.google.android.gms:play-services-location:11.8.0'
implementation
   'com.google.android.gms:play-services-maps:11.8.0'
```

Last Known Location

The easiest way to get the device's position is to get the *last known location*. To do so, in your app request permissions, add the following:

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

GPS resolution with fewer than ten yards needs `FINE` location permission, while the coarser network-based resolution with approximately 100 yards needs the `COARSE` location permission. Adding both to the manifest file gives us the most options, but depending on your needs, you could continue with just the coarse one.

Then, inside you component (for example, inside `onCreate()`), you construct a `FusedLocationProviderClient` as follows:

```
var fusedLocationClient: FusedLocationProviderClient? =
    null

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

  ...

  fusedLocationClient = LocationServices.
         getFusedLocationProviderClient(this)
}
```

Wherever needed in your app, you can use it to get the last known location.

```
if (checkPermission(
       Manifest.permission.ACCESS_COARSE_LOCATION,
       Manifest.permission.ACCESS_FINE_LOCATION)) {
```

```
  fusedLocationClient?.lastLocation?.
          addOnSuccessListener(this,
      {location : Location? ->
          // Got last known location. In some rare
          // situations this can be null.
          if(location == null) {
              // TODO, handle it
          } else location.apply {
              // Handle location object
              Log.e("LOG", location.toString())
          }
      })
}
```

Here, `checkPermission()` checks and possibly acquires the needed permissions as described in Chapter 7. This could be, for example, the following:

```
val PERMISSION_ID = 42
private fun checkPermission(vararg perm:String) :
      Boolean {
  val havePermissions = perm.toList().all {
      ContextCompat.checkSelfPermission(this,it) ==
            PackageManager.PERMISSION_GRANTED
  }
  if (!havePermissions) {
    if(perm.toList().any {
      ActivityCompat.
      shouldShowRequestPermissionRationale(this, it)}
    ) {
        val dialog = AlertDialog.Builder(this)
            .setTitle("Permission")
            .setMessage("Permission needed!")
            .setPositiveButton("OK",{
                id, v ->
                ActivityCompat.requestPermissions(
                    this, perm, PERMISSION_ID)
            })
            .setNegativeButton("No",{
                 id, v ->
            })
            .create()
        dialog.show()
    } else {
      ActivityCompat.requestPermissions(this, perm,
          PERMISSION_ID)
    }
    return false
  }
  return true
}
```

For simplicity I used strings for button labels and messages. For production code, make sure you use resources! The function `checkPermission()`, if necessary, tries to acquire permission from a system activity. Whether the user grants permissions, upon return from this activity, your app may accordingly react to the result.

```
override
fun onRequestPermissionsResult(requestCode: Int,
         permissions: Array<String>,
         grantResults: IntArray) {
   when (requestCode) {
     PERMISSION_ID -> {
         ...
     }
     ...
   }
}
```

> **Caution** The concept of a "last known location" is somewhat blurry. In an emulated device, for example, it is not sufficient to set the location by the provided device control for changing the last known location. Only after an app like Google Maps uses location update mechanisms does the code described here return the correct value. The mechanisms described in the following sections are more complex but also more reliable.

Tracking Position Updates

If your app needs to track updates on changing locations, you follow a different approach. First, the permissions needed are the same as stated earlier for the last known location, so there's no change there. The difference is in requesting periodic updates from the *fused location provider*. For that we need to define a location settings object. Confusingly, the corresponding class is called `LocationRequest` (it would have been better as `LocationRequestSettings` or something else). To create one, write the following:

```
val reqSetting = LocationRequest.create().apply {
      fastestInterval = 10000
      interval = 10000
      priority = LocationRequest.PRIORITY_HIGH_ACCURACY
      smallestDisplacement = 1.0f
}
```

The `.apply` construct lets us configure the object faster. For example, `fastestInterval = 10000` internally gets translated to `reqSetting.setFastestInterval(10000)`. The meanings of the individual settings are as follows:

- `fastestInterval`

  The fastest possible total update interval in milliseconds for the location provider.

- `interval`

The requested interval in milliseconds. This setting is only approximate.

- `priority`

  The requested accuracy. This setting influences the battery usage. The following are possible values (constants from `LocationRequest`):

  - `PRIORITY_NO_POWER`: Fetches passive updates only if other requestors actively request updates
  - `PRIORITY_LOW_POWER`: Updates only on "city" levels
  - `PRIORITY_BALANCED_POWER_ACCURACY`: Updates only on "city street block" levels
  - `PRIORITY_HIGH_ACCURACY`: Uses the highest possible accuracy available

  This value will be internally adapted according to available permissions.

- `smallestDisplacement`

  This is the smallest displacement in meters necessary for an update message to be fired.

With that location request setting, we check whether the system is able to fulfill our request. This happens in the following code snippet:

```
val REQUEST_CHECK_STATE = 12300 // any suitable ID
val builder = LocationSettingsRequest.Builder()
        .addLocationRequest(reqSetting)
val client = LocationServices.getSettingsClient(this) client.checkLocationSettings(builder.
build()).
        addOnCompleteListener { task ->
    try {
        val state: LocationSettingsStates = task.result.
               locationSettingsStates
        Log.e("LOG", "LocationSettings: \n" +
          " BLE present: ${state.isBlePresent} \n" +
          " BLE usable: ${state.isBleUsable} \n" +
          " GPS present: ${state.isGpsPresent} \n" +
          " GPS usable: ${state.isGpsUsable} \n" +
          " Location present: " +
               "${state.isLocationPresent} \n" +
          " Location usable: " +
               "${state.isLocationUsable} \n" +
          " Network Location present: " +
               "${state.isNetworkLocationPresent} \n" +
          " Network Location usable: " +
               "${state.isNetworkLocationUsable} \n"
        )
    } catch (e: RuntimeExecutionException) {
        if (e.cause is ResolvableApiException)
            (e.cause as ResolvableApiException).
              startResolutionForResult(
```

```
                    this@MainActivity,
                    REQUEST_CHECK_STATE)
    }
}
```

This asynchronously performs a check. If high accuracy is requested and the device's setting won't allow updates based on GPS data, the corresponding system settings dialog gets called. The latter happens somewhat awkwardly inside the exception catch. The result of the corresponding system intent call ends up in the following:

```
override fun onActivityResult(requestCode: Int,
      resultCode: Int, data: Intent) {
  if (requestCode and 0xFFFF == REQUEST_CHECK_STATE) {
      Log.e("LOG", "Back from REQUEST_CHECK_STATE")
      ...
  }
}
```

With all set up correctly and enough permissions, we now can register for location updates via the following:

```
val locationUpdates = object : LocationCallback() {
    override fun onLocationResult(lr: LocationResult) {
        Log.e("LOG", lr.toString())
        Log.e("LOG", "Newest Location: " + lr.locations.
 last())
        // do something with the new location...
    }
}
fusedLocationClient?.requestLocationUpdates(reqSetting,
      locationUpdates,
      null /* Looper */)
```

To stop location updates, you move `locationUpdates` to a class field and react to stop requests via the following:

```
fun stopPeriodic(view:View) {
  fusedLocationClient?.
        removeLocationUpdates(locationUpdates)
}
```

Geocoding

The `Geocoder` class allows you to determine the geocoordinates (longitude, latitude) for a given address or, conversely, possible addresses for given geocoordinates. These processes are known as *forward* and *reverse geocoding*. The `Geocoder` class internally uses an online Google service, but the details are hidden inside the implementation. You as a developer can use the `Geocoder` class without the need to understand where the data come from.

This section is about *reverse* geocoding. We use a `Location` object with longitude and latitude to find nearby street names. To start, we first have to decide what we do with a potentially long-running operation. To do the lookup, a network operation is necessary, and

the online service needs to look up a huge database. An `IntentService` will do the job for us, and from among the methods that can return us the value, we choose a `ResultReceiver` passed by intent extras. First we define kind of a contract between the service and the service clients of a class holding some constants.

```
class GeocoderConstants {
  companion object Constants {
      val SUCCESS_RESULT = 0
      val FAILURE_RESULT = 1
      val PACKAGE_NAME = "<put your package name here>"
      val RECEIVER = "$PACKAGE_NAME.RECEIVER"
      val RESULT_DATA_KEY =
          "$PACKAGE_NAME.RESULT_DATA_KEY"
      val LOCATION_DATA_EXTRA =
          "$PACKAGE_NAME.LOCATION_DATA_EXTRA"
  }
}
```

Now the full service class reads as follows.

```
class FetchAddressService :
      IntentService("FetchAddressService") {
      override
      fun onHandleIntent(intent: Intent?) {
          val geocoder = Geocoder(this, Locale.getDefault())
          var errorMessage = ""

          // Get the location passed to this service through
          // an extra.
          val location = intent?.getParcelableExtra(
                   GeocoderConstants.LOCATION_DATA_EXTRA)
              as Location

          // Get the Intent result receiver
          val receiver = intent.getParcelableExtra(
              GeocoderConstants.RECEIVER) as ResultReceiver

          var addresses: List<Address>? = null
          try {
              addresses = geocoder.getFromLocation(
                      location.getLatitude(),
                      location.getLongitude(),
                      1) // Get just a single address!
          } catch (e: IOException) {
              // Catch network or other I/O problems.
               errorMessage = "service_not_available"
               Log.e("LOG", errorMessage, e)
          } catch (e: IllegalArgumentException) {
              // Catch invalid latitude or longitude values.
               errorMessage = "invalid_lat_long_used"
               Log.e("LOG", errorMessage + ". " +
                    "Latitude = " + location.getLatitude() +
```

```
                    ", Longitude = " +
                     location.getLongitude(), e)
        }

        if (addresses == null || addresses.size == 0) {
            // No address was found.
            if (errorMessage.isEmpty()) {
                 errorMessage = "no_address_found"
                 Log.e("LOG", errorMessage)
            }
            deliverResultToReceiver(
            receiver,
             GeocoderConstants.FAILURE_RESULT,
              errorMessage)
    } else {
        val address = addresses[0]
        val addressFragments =
               (0..address.maxAddressLineIndex).
              map { i -> address.getAddressLine(i) }
        val addressStr = addressFragments.joinToString(
               separator =
                   System.getProperty("line.separator"))
         Log.i("LOG", "address_found")
         deliverResultToReceiver(
              receiver,
               GeocoderConstants.SUCCESS_RESULT,
               addressStr)
    }
  }

  private fun deliverResultToReceiver(
        receiver:ResultReceiver,
        resultCode: Int,
        message: String) {
   val bundle = Bundle()
   bundle.putString(GeocoderConstants.RESULT_DATA_KEY,
        message)
   receiver.send(resultCode, bundle)
   }
}
```

Again, for productive code, you should use resource strings instead of literals, as shown in this example. The service must be registered inside `AndroidManifest.xml`.

```
<service android:name=".FetchAddressService"
         android:exported="false"/>
```

For using this service, we first build a `ResultReceiver` class and check the permissions. For example, we use the last known location to call the service.

```
class AddressResultReceiver(handler: Handler?) :
      ResultReceiver(handler) {
  override
```

```
  fun onReceiveResult(resultCode: Int,
                      resultData: Bundle) {
    val addressOutput =
              resultData.getString(
                   GeocoderConstants.RESULT_DATA_KEY)
          Log.e("LOG", "address result = " +
                 addressOutput.toString())
          ...
    }
  }
  val resultReceiver = AddressResultReceiver(null)
  fun startFetchAddress(view:View) {
    if (checkPermission(
          Manifest.permission.ACCESS_COARSE_LOCATION,
          Manifest.permission.ACCESS_FINE_LOCATION))
    {
        fusedLocationClient?.lastLocation?.
            addOnSuccessListener(this, {
                 location: Location? ->
               if (location == null) {
                 // TODO
               } else location.apply {
                  Log.e("LOG", toString())
                  val intent = Intent(
                     this@MainActivity,
                       FetchAddressService::class.java)
                 intent.putExtra(
                   GeocoderConstants.RECEIVER,
                    resultReceiver)
                 intent.putExtra(
                   GeocoderConstants.LOCATION_DATA_EXTRA,
                    this)
                 startService(intent)
               }
        })
  }
}
```

You can see we use an explicit intent; that is why we don't need an intent filter inside the service declaration in `AndroidManifest.xml`.

Using ADB to Fetch Location Information

For development and debugging purposes, you can use ADB to fetch the location information of a device connected to your PC or laptop.

```
./adb shell dumpsys location
```

For more information on CLI commands, see Chapter 18.

Maps

Adding a map to your location-related app greatly improves the usability for your users. To add a Google API map, the easiest way is to use the wizard provided by Android Studio. Follow these steps:

1. Add a map activity: right-click your module, in the menu choose New ➤ Activity ➤ Gallery, and from the gallery choose Google Maps Activity. Click Next. On the screen that follows, enter activity parameters according to your needs. However, choosing an appropriate activity name basically is all you need. The defaults make sense in most cases.

2. You need an API key to use Google Maps. For this purpose, inside the file `/res/values/google_maps_api.xml`, locate the link inside the comments; it might look like `https://console.developers.google.com/flows/enableapi?...`. Open this link in a browser and follow the instructions there. Finally, enter the key generated online as the text of the `<string name = "google_maps_key" ... >` element in that file.

What we have now is an activity class prepared for us, a fragment layout file that we can include in our app, and a registered API key that allows us to fetch maps data from the Google server.

To include the fragment as defined by `/res/layout/activity_maps.xml`, you write the following in your layout, with sizes adapted according to your needs:

```
<include
    android:layout_width="fill_parent"
    android:layout_height="250dp"
    layout="@layout/activity_maps" />
```

Inside the code we first add a snippet to fetch a map from the server. You do this inside your `onCreate()` callback as follows:

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    val mapFragment = supportFragmentManager
            .findFragmentById(R.id.map)
        as SupportMapFragment
    mapFragment.getMapAsync(this)
}
```

Here, `R.id.map` points to the map's ID inside `/res/layout/activity_maps.xml`.

Next we add a callback that gets called when the map is loaded and ready to receive commands. To do so, we extend the activity that handles the map to also implement the interface `com.google.android.gms.maps.OnMapReadyCallback`.

```
class MainActivity : AppCompatActivity(),
      OnMapReadyCallback { ... }
```

Add the callback implementation. Here's an example:

```
/**
  * Use this to save and manipulate the map once
  * available.
  */
 override fun onMapReady(map: GoogleMap) {
     // Add a marker in Austin, TX and move the camera
     val austin = LatLng(30.284935, -97.735464)
     map.addMarker(MarkerOptions().position(austin).
         title("Marker in Austin"))
     map.moveCamera(CameraUpdateFactory.
        newLatLng(austin))
 }
```

If Google Play Services is not installed on the device, the user automatically gets prompted to install it. The map object can, of course, be saved as a class object field, and you can do lots of interesting things with it, including adding markers, lines, zoom, movement, and more. The possibilities are described in the online API documentation for `com.google.android.gms.maps.GoogleMap`.

Preferences

Preferences allow the user to change the way the app performs certain parts of its functionalities. Contrary to the input given by the user during the app's functional workflows, preferences are less likely to be changed, so the access to preferences is usually provided by a single preferences entry in the app's menu.

You as a developer might decide to develop special activities for preferences from scratch, but you don't have to do this. In fact, the Preferences API provided by the Android OS is quite versatile and allows you to present a preferences or settings dialog in a standard way, letting your app appear more professional. Also, you don't have to implement the GUI yourself.

To start with an example preferences workflow, create a class like this:

```
class MySettingsFragment : PreferenceFragment(),
    SharedPreferences.OnSharedPreferenceChangeListener {
  companion object {
      val DELETE_LIMIT = "pref_key_delete_limit"
      val LIST = "pref_key_list"
      val RINGTONE = "pref_key_ringtone"
  }

  override fun onSharedPreferenceChanged(
         sharedPreferences: SharedPreferences?,
         key: String?) {
    sharedPreferences?.run {
        when(key) {
           DELETE_LIMIT -> {
            findPreference(key).summary =
                   getString(key, "") ?: "10"
           }
```

```
            LIST -> {
              findPreference(key).summary =
                    (findPreference(key) as ListPreference).
                    entry
            }
            RINGTONE -> {
              val uriStr = getString(key, "") ?: ""
              findPreference(key).summary =
                      getRingtoneName(Uri.parse(uriStr))
            }
        }
    }
}

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Load the preferences from an XML resource
    addPreferencesFromResource(R.xml.preferences)

    val sharedPref = PreferenceManager.
           getDefaultSharedPreferences(activity)
    sharedPref.registerOnSharedPreferenceChangeListener(
          this)

    with(sharedPref) {
      findPreference(DELETE_LIMIT).summary =
          getString(DELETE_LIMIT, "10")
      findPreference(LIST).summary =
          (findPreference(LIST) as ListPreference).let {
              val ind = Math.max(0, it.findIndexOfValue(
  it.value))
              resources.getStringArray(listentries)[ind]
           }
      findPreference(RINGTONE).summary =
          getRingtoneName(
                Uri.parse(getString(RINGTONE, "") ?: ""))
    }
  }

  fun getRingtoneName(uri:Uri):String {
      return activity.contentResolver.
              query(uri, null, null, null, null)?.let {
           it.moveToFirst()
           val res = it.getString(
                  it.getColumnIndex(
                  MediaStore.MediaColumns.TITLE))
            it.close()
            res
      } ?: ""
  }
}
```

This defines a fragment that has the following parameters:

- On creation, it sets the preferences resource, as described in a moment.
- On creation, it accesses the preferences API using the `PreferenceManager`. We use it to set a preferences change listener, so we can update the UI when preferences change. In addition, we fetch a couple of preferences to prepare the preferences UI and set "summary" lines that get shown inside preference items.
- The listener `onSharedPreferenceChanged()` is used to update the summary lines for applicable preferences.

The actual preferences layout gets defined inside `res/xml/`, in this case, as a resources file `preferences.xml` with these contents:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android=
      "http://schemas.android.com/apk/res/android">
  <CheckBoxPreference
      android:key="pref_key_auto_delete"
      android:summary="Auto delete"
      android:title="Auto delete"
      android:defaultValue="false" />
  <EditTextPreference
      android:key="pref_key_delete_limit"
      android:dependency="pref_key_auto_delete"
      android:summary="Delete Limit"
      android:title="Delete Limit"
      android:defaultValue="10" />
  <ListPreference android:key="pref_key_list"
      android:summary="A List"
      android:title="A List"
      android:entries="@array/listentries"
      android:entryValues="@array/listvalues"
      android:defaultValue="1" />
  <MultiSelectListPreference
      android:key="pref_key_mslist"
      android:summary="A Multiselect List"
      android:title="A Multiselect List"
      android:entries="@array/listentries"
      android:entryValues="@array/listvalues"
      android:defaultValue="@array/multiselectdefaults"/>
  <SwitchPreference
      android:key="pref_key_switch"
      android:summary="A Switch"
      android:title="A Switch"
      android:defaultValue="false" />
  <RingtonePreference
      android:key="pref_key_ringtone"
      android:summary="A Ringtone"
      android:title="A ringtone"
      />
</PreferenceScreen>
```

The keys used here must match the key strings used inside the fragment defined earlier. Note that for simplicity I used plain strings in that file. For a productive environment, you should of course refer to resource strings.

In addition, in `res/values` add a file `arrays.xml` and in it write the following for the various arrays we use in the example:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="listentries">
      <item>List item 1</item>
      <item>List item 2</item>
      <item>List item 3</item>
  </string-array>
  <string-array name="listvalues">
      <item>1</item>
      <item>2</item>
      <item>3</item>
  </string-array>
  <string-array name="multiselectdefaults">
      <item>1</item>
      <item>3</item>
  </string-array>
</resources>
```

What is left is a place in your app where to insert preferences. You could, for example, add the following at a suitable place:

```
<FrameLayout
    android:id="@+id/prefsFragm"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

In the app call, start the preferences workflow.

```
fragmentManager.beginTransaction()
         .replace(prefsFragm.id, MySettingsFragment())
         .commit()
```

For the various default values we defined in `preferences.xml` to take effect, you must at any place in your app where preferences get accessed first call the following:

```
PreferenceManager.setDefaultValues(
      this, preferences.id, false)
```

This could be done in an activity's `onCreate()` callback. Figure 8-9 shows this preferences example.

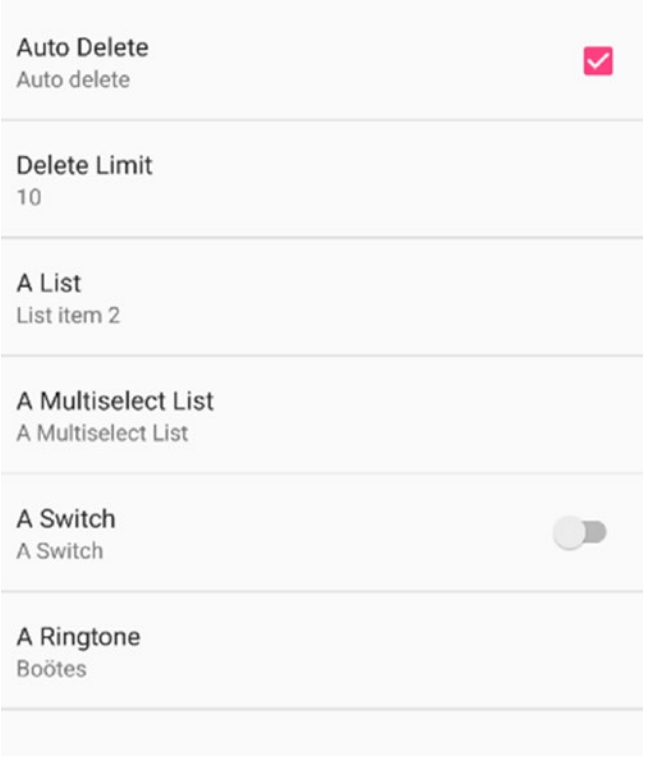

Figure 8-9. *A preferences screen*

For more settings, the settings screen as designed so far might be a little hard to read. To add some structure, you can gather setting items in groups and add group titles. This is done in the `preferences.xml` file as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android=
      "http://schemas.android.com/apk/res/android">
  ...
  <PreferenceCategory
        android:title="Category Title">
      <SwitchPreference
          android:key="pref_key_switch"
          android:summary="A Switch"
          android:title="A Switch"
          />
      <RingtonePreference
          android:key="pref_key_ringtone"
          android:summary="A Ringtone"
          android:title="A Ringtone"
          />
  </PreferenceCategory>
  ...
```

You can instead have the entries open a preferences subscreen. For this aim, again in `preferences.xml` you write the following:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android=
      "http://schemas.android.com/apk/res/android">
  ...
  <PreferenceScreen
        android:title="Subscreen Title"
        android:persistent="false">
      <SwitchPreference
               android:key="pref_key_switch"
               android:summary="A Switch"
               android:title="A Switch"
              />
      <RingtonePreference
           android:key="pref_key_ringtone"
           android:summary="A Ringtone"
           android:title="A Ringtone"
          />
  </PreferenceScreen>
  ...
```

Advanced preferences features include a standardized header-contents way of presenting preferences, overwriting the setting item UIs, creating your own preferences UI, and tweaking the preferences data location. For details, refer to the online documentation of the Settings API.

Chapter 9

User Interface

The user interface is certainly the most important part of any end-user app. For corporate usage, apps without user interfaces are possible, but even then in most cases you will have some kind of rudimentary UI, if for no other reason just to avoid the Android OS killing your app too readily during resource housekeeping tasks.

In this chapter, we will not cover the basics of UI development for Android. Instead, it is expected that you've read some introductory-level book or worked through the official tutorials (or any number of the other tutorials you will find on the Web). What we do here is to cover a couple of important UI-related issues that help you to create more stable apps or apps with special outstanding requirements.

Background Tasks

Android relies on a single-threaded execution model. This means when an app starts, it by default starts only a single thread, called the *main thread*, in which all actions run unless you explicitly use background threads for certain tasks. This automatically means you have to take special precautions if you have long-running tasks that would interrupt a fluent UI workflow. It is not acceptable for modern apps to have the UI freeze after the user taps a button or because this action leads to a process running for a few seconds or longer. It is therefore vital to put long-running tasks into the background.

One way to accomplish putting things into the background is to use `IntentService` objects, as described in Chapter 4. Depending on the circumstances, it might, however, blow up your app design to put all background work into services; in addition, having too many services run on a device will not help keep resources usage low. Another option is to use loaders as described in Chapter 8. For low-level tasks, however, it is better to use a more low-level approach. You have several options here, which we describe in the following sections.

P. Späth, *Pro Android with Kotlin,* https://doi.org/10.1007/978-1-4842-3820-2_9

Java Concurrency

At a low level, you can use Java threads and classes from the `java.util.concurrent` package to handle background jobs. Beginning with Java 7, those classes have become quite powerful, but you need to fully understand all options and implications.

You will quite often read that directly handling threads from inside the Android OS is not a good idea because threads are expensive when speaking of system resources. While this was certainly true for older devices and old Android versions, nowadays this is just no longer the case. For me, a simple test on a Motorola Moto G4 starting 1,000 threads and waiting until all are running took approximately 0.0006 seconds per thread. So, if you are used to Java threads and less than a millisecond for the thread to start is good for you, there is no performance reason for not using Java threads. However, you must take into account that threads run outside any Android component lifecycle, so you have to handle lifecycle issues manually if you use threads.

In Kotlin, threads are defined and started easily.

```
val t = Thread{ ->
  // do background work...
}.also { it.start() }
```

Note that accessing the UI from inside a thread is not allowed in Android. You must do that as follows:

```
val t = Thread{ ->
  // do background work...
  runOnUiThread {
    // use the UI...
  }
}.also { it.start() }
```

The AsyncTask Class

An `AsyncTask` object is a medium-level helper class to run some code in the background. You override its `doInBackground()` method to do some background work, and if you need to communicate with the UI, you also implement `onProgressUpdate()` to do the communication and fire `publishProgress()` from inside the background work to trigger it.

> **Note** Android Studio will as a warning complain about a possible memory leak if you create `AsyncTask` objects like `val at = object : AsyncTask< Int, Int, Int >() { ... }`. This is because internally a static reference to the background code will be held. The warning can be suppressed by annotating the method with `@SuppressLint("StaticFieldLeak")`.

> **Caution** A number of `N AsyncTask` jobs will not lead to a parallel execution of all `N` of them. Instead, they all run sequentially in *one* background thread.

Handlers

A Handler is an object maintaining a message queue for the asynchronous processing of messages or Runnable objects. You can use a Handler for asynchronous processes as follows:

```
var handlerThread : HandlerThread? = null
// or: lateinit var handlerThread : HandlerThread
...
fun doSomething() {
  handlerThread = handlerThread ?:
        HandlerThread("MyHandlerThread").apply {
    start()
  }
  val handler = Handler(handlerThread?.looper)

  handler.post {
    // do s.th. in background
  }
}
```

If you create one HandlerThread as in this code snippet, everything that is posted gets run in the background, but it is executed sequentially there. This means handler.post{}; handler.post{} will run the posts in series. You can, however, create more HandlerThread objects to handle the posts in parallel. For true parallelism, you'd have to use one HandlerThread for each execution.

> **Note** Handlers were introduced in Android a long time before the new java.util.concurrent package was available in Java 7. Nowadays for your own code, you might decide to favor the generic Java classes over Handlers without missing anything. Handlers, however, quite often show up in Android's libraries.

Loaders

Loaders also do their work in the background. They are primarily used for loading data from an external source. Loaders are described in Chapter 8.

Supporting Multiple Devices

Device compatibility is an important issue for apps. Whenever you create an app, it is of course your goal to address as many device configurations as possible and to make sure users who install your app on a certain device can actually use it. Compatibility boils down to the following:

- Finding a way your app can run with different screen capabilities, including pixel width, pixel height, pixel density, color space, and screen shape
- Finding a way your app can run with different API versions

- Finding a way your app can run with different hardware features, including sensors and keyboards
- Finding a way you can filter your app's visibility inside the Google Play store
- Possibly providing different APKs for one app, depending on device features

In this chapter, we talk about UI-related issues for compatibility; we focus on screen and user input capabilities.

Screen Sizes

To allow your app to look nice on different screen sizes, you can do the following:

- **Use flexible layouts**

  Avoid specifying absolute positions and absolute width. Instead, use layouts that allow specifications like "on the right of" or "use half of the available space" or similar.

- **Use alternative layouts**

  Using alternative layouts is a powerful means for supplying different screen sizes. The layout XML files can be put into different directories with names containing size filters. For example, you could put one layout into the file `res/layout/main_activity.xml` and another one into `res/layout-sw480dp/main_activity.xml` expressing a "smallest width" of 480dp (for large phone screens 5" or higher). The naming schema gets extensively described online in the "providing resources" and "providing alternative resources" documents in the Android developer documentation. The Android OS then automatically picks the best matching layout during runtime on the user's device.

- **Use stretchable images**

  You can provide *nine-patch* bitmaps for UI elements. Inside such images you provide a 1-pixel-wide border telling which parts of an image can be repeated to stretch an image and optionally which parts can be used for inner contents. Such nine-patch images are PNG files with the suffix `.9.png`. Android Studio allows for converting conventional PNGs into nine-patch PNGs; use the context menu for that purpose.

Pixel Densities

Devices have different pixel densities. To make your app as device independent as possible, wherever you need pixel sizes, use *density-independent pixel* sizes instead of *pixel sizes*. Density-independent pixel sizes use dp as a unit, while pixels use px.

In addition, you can also provide different layout files based on different densities. The separation is similar to the separation for different screen sizes described earlier.

Declare Restricted Screen Support

In some situations, you want to restrict your app by saying that some screen characteristics just cannot be used. Obviously, you want to avoid such situations, but in case it is inevitable, you can do so in `AndroidManifest.xml`.

- Tell the app that certain activities can run in multiwindow modes available on API level 24 (Android 7.0) or later. For that aim, use attribute `android:resizeableActivity` and set it to `true` or `false`.
- Tell certain activities they should be letter-boxed (have appropriate margins added) above certain aspect ratios. For that aim, use attribute `android:maxAspectRatio` and specify the aspect ratio as a value. For Android 7.1 and older, duplicate this setting in the `<application>` element, as in `<meta-data android:name = "android.max_aspect" android:value = "s.th." />`.
- Tell certain activities they should not be stretched above a certain limit by using the `largestWidthLimitDp` attribute inside a `<supports-screens>` element.
- Use more `<supports-screens>` and `<compatible-screens>` elements and attributes, as described in Chapter 2.

Detect Device Capabilities

From inside your app, you can check for certain features as follows:

```
if(packageManager.
      hasSystemFeature(PackageManager.FEATURE_...)) {
  // ...
}
```

Here, `FEATURE_...` is one of the various constants from inside `PackageManager`.

An additional source of feature information is `Configuration`. In Kotlin, from inside an activity, you can obtain the configuration object via the following:

```
val conf = resources.configuration
```

From there, obtain information about the color mode in use, available keyboards, screen orientation, and touchscreen capabilities. To get the screen's size, you can write the following:

```
val size = Point()
windowManager.defaultDisplay.getSize(size)
// or (getSystemService(Context.WINDOW_SERVICE)
//    as WindowManager).defaultDisplay.getSize(size)
val (width,height) = Pair(size.x, size.y)
```

To get the resolution, you can use the following:

```
val metrics = DisplayMetrics()
windowManager.defaultDisplay.getMetrics(metrics)
val density = metrics.density
```

Programmatic UI Design

Usually UI design happens by declaring UI objects (`View` objects) and containers (`ViewGroup` objects) inside one or more XML layout files. While this is the suggested way of designing a UI and most people probably tell you shouldn't do anything else, there are reasons to take away the layout design from XML and do a programmatical layout instead.

- You need more dynamics on a layout. For example, your app adds, removes, or moves layout elements by user actions. Or you want to create a game with game elements represented by `View` objects that are dynamically moving, appearing, and disappearing.
- Your layout is defined on a server. Especially in a corporate environment, defining the layout on a server makes sense. Whenever the layout of an app changes, you don't need a new version to be installed on all devices. Instead, only a central layout engine needs to be updated.
- You define a layout builder that allows to specify layouts in Kotlin in a more expressive and concise way compared to XML. Take a look at the following, for example, which is valid Kotlin syntax:

  ```
  LinearLayout(orientation="vertical") {
    TextView(id="tv1",
          width="match_parent", height="wrap_content")
    Button(id="btn1", text="Go",
          onClick = { btn1Clicked() })
  }
  ```

- You need special constructs that are not defined in XML. While the standard way is to define everything in XML as much as possible and do the rest in the code, you might prefer a single-technology solution, and in that case you have to do everything from inside the code.

Note that if you abandon the descriptive layout via XML files and use a programmatic layout instead, you manually have to take care of different screen sizes, screen densities, and other hardware characteristics. While this is always possible, under certain circumstances it could be a complicated and error-prone procedure. Certain characteristics such as UI element sizes can be much more easily expressed in XML than in the code.

To start with a programmatic UI design, it is the easiest if you define a single container inside XML and use it in your code. I say this because layouts have their own idea of how and when to place their children, so you might end up in a nightmare of timing, layout, and clipping issues if your code has another idea about how and when to place views. A good candidate is a `FrameLayout`, as shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
  xmlns:android =
        "http://schemas.android.com/apk/res/android"
  android:id="@+id/fl"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
</FrameLayout>
```

Use this as a layout XML file, say `/res/layout/activity_main.xml`, and write the following as a sample activity:

```
class MainActivity : AppCompatActivity() {
  var tv:TextView? = null
  override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)

      // For example add a text at a certain position
      tv = TextView(this).apply {
          text = "Dynamic"
          x = 37.0f
          y = 100.0f
      }
     fl.addView(tv)
  }
}
```

To add a button that shifts the text from the previous example around, you can write the following:

```
val WRAP = ViewGroup.LayoutParams(
                ViewGroup.LayoutParams.WRAP_CONTENT,
                ViewGroup.LayoutParams.WRAP_CONTENT)
fl.addView(
      Button(this).apply {
        text = "Go"
        setOnClickListener { v ->
          v?.run {
            x += 30.0f *
                (-0.5f + Math.random().toFloat())
            y += 30.0f *
                (-0.5f + Math.random().toFloat())
           }
        }
      }, WRAP
)
```

If you don't need total control and want a layout object to do its children positioning and sizing job the way it is designed, adding children to other layout types like, for example, a `LinearLayout`, this is possible from inside the code. Just use the `addView()` method without explicitly setting the position via `setX()` or `setY()`. Under certain circumstances you must

use `layoutObject.invalidate()` to trigger a re-layout afterward. The latter has to be done from inside the UI thread or inside `runOnUiThread{ ... }`.

Adapters and List Controls

The need to display a list with a variable number of items happens quite often, especially in a corporate environment. While `AdapterView` and `Adapter` objects with various subclasses have been around for a while, we concentrate on the relatively new and high-performing *recycler views*. You will see that with Kotlin's conciseness, implementing a recycler view happens in an elegant and comprehensive manner.

The basic ides is as follows: you have an array or a list or another collection of data items, maybe from a database, and you want to send them to a single UI element that does all the presentation, including rendering all visible items and providing a scroll facility if necessary. Each item's presentation either should depend on an *item* XML layout file or be generated dynamically from inside the code. The mapping from each data item's member to the corresponding view element from inside the item's UI representation is to be handled by an *adapter* object.

With recycler views, this all happens in a straightforward manner, but first we have to include a support library because the recycler views are not part of the framework. To do so, inside your *module*'s `build.gradle` file, add the following:

```
implementation
      'com.android.support:recyclerview-v7:26.1.0'
```

Add this inside the `dependencies{ ... }` section (on one line; remove the newline after `implementation`).

To tell the app we want to use a recycler view, inside your activity's layout file, add the following:

```
<android.support.v7.widget.RecyclerView
    android:id="@+id/recycler_view"
    android:scrollbars="vertical"
       ... />
```

Specify its layout options as for any other `View`.

For the layout of an item from the list, create another layout file inside `res/layout`, say `item.xml`, with the following sample content:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/
   apk/res/android"
                android:layout_width="fill_parent"
                android:layout_height="?android:attr/
   listPreferredItemHeight"
                android:padding="8dip" >
```

```
<ImageView
    android:id="@+id/icon"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:layout_alignParentBottom="true"
    android:layout_alignParentTop="true"
    android:layout_marginRight="8dip"
    android:contentDescription="TODO"
    android:src="@android:drawable/star_big_on" />

<TextView
    android:id="@+id/secondLine"
    android:layout_width="fill_parent"
    android:layout_height="26dip"
    android:layout_alignParentBottom="true"
    android:layout_alignParentRight="true"
    android:layout_toRightOf="@id/icon"
    android:singleLine="true"
    android:text="Description"
    android:textSize="12sp" />

<TextView
    android:id="@+id/firstLine"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_above="@id/secondLine"
    android:layout_alignParentRight="true"
    android:layout_alignParentTop="true"
    android:layout_alignWithParentIfMissing="true"
    android:layout_toRightOf="@id/icon"
    android:gravity="center_vertical"
    android:text="Example application"
    android:textSize="16sp" />
</RelativeLayout>
```

As stated earlier, you could also omit this step and define an item's layout solely from inside the code! Next we provide an adapter. In Kotlin this could be as easy as follows:

```
class MyAdapter(val myDataset:Array<String>) :
      RecyclerView.Adapter
            <MyAdapter.Companion.ViewHolder>() {
  companion object {
      class ViewHolder(val v:RelativeLayout) :
            RecyclerView.ViewHolder(v)
  }

  override
  fun onCreateViewHolder(parent:ViewGroup,
          viewType:Int) : ViewHolder {
      val v = LayoutInflater.from(parent.context)
              .inflate(R.layout.item, parent, false)
            as RelativeLayout
      return ViewHolder(v)
  }
```

```
  override
  fun onBindViewHolder(holder:ViewHolder,
                       position:Int) {
      // replace the contents of the view with
      // the element at this position
      holder.v.findViewById<TextView>(
            R.id.firstLine).text =
            myDataset[position]
  }

  override
  fun getItemCount() : Int = myDataset.size
}
```

Here are a couple of notes on that listing:

- The class inside the "companion object" is Kotlin's way of declaring a static inner class. This one designates a reference to each data item as a UI element. More precisely, the recycler view will internally hold only so many view holders as are necessary to represent the *visible* items.
- Only when really needed the function `onCreateViewHolder()` to create a view holder gets called. More precisely, it's called more or less only as often as is necessary to render the items visible to the user.
- The function `onBindViewHolder()` connects one of the visible view holders with a certain data item. Here we must replace the contents of a view holder's view.

Inside the activity all that is needed to define the recycler view is the following:

```
with(recycler_view) {
  // use this setting to improve performance if you know
  // that changes in content do not change the layout
  // size of the RecyclerView
  setHasFixedSize(true)
  // use for example a linear layout manager
  layoutManager = LinearLayoutManager(this@MainActivity)
  // specify the adapter, use some sample data
  val dataset = (1..21).map { "Itm" + it }.toTypedArray()
  adapter = MyAdapter(dataset)
}
```

This will look like Figure 9-1. The following are useful extensions to the program:

- Add on-click listeners to all items
- Make items selectable
- Make items or item parts editable
- Automatically react to changes in the underlying data
- Tailor graphical transition effects

For all that I refer to the online documentation of recycler views. The code presented here, however, should give you a good starting point.

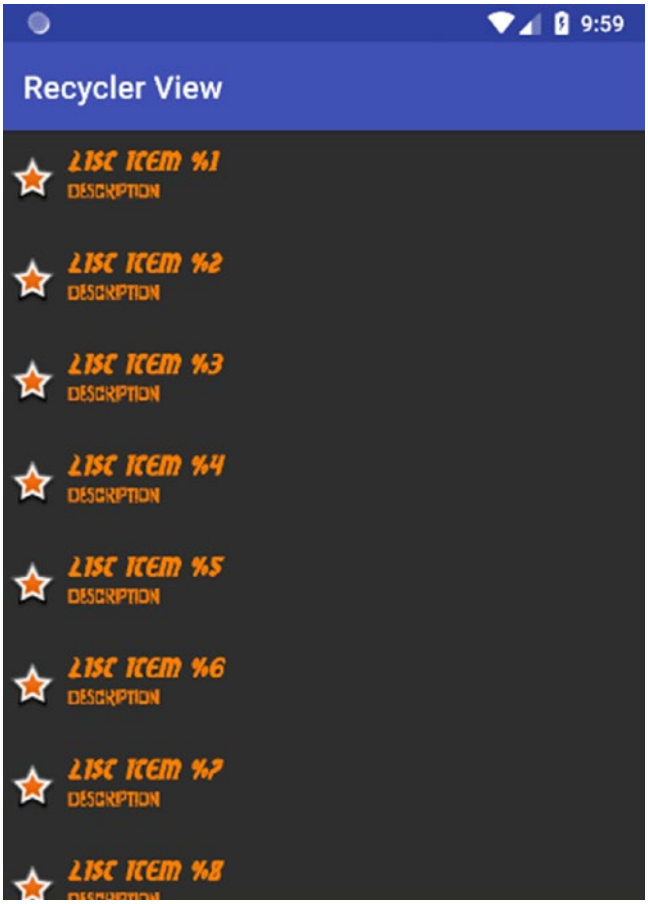

Figure 9-1. Recycler view

Styles and Themes

The predefined styles an Android app uses by default already give a good starting point for professional-looking apps. If, however, you want to apply your company's style guidelines or otherwise create a visually outstanding app, creating your own styles is worth the effort. Even better, create your own theme, which is a collection of styles to be applied to groups of UI elements.

Styles and themes get created inside `res/values/` as XML files. To make a new theme, you use or create a file called `themes.xml` and write something like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="MyTheme" parent="Theme.AppCompat">
      <item name="colorPrimary">
            @color/colorPrimary</item>
```

```
      <item name="colorPrimaryDark">
            @color/colorPrimaryDark</item>
      <item name="colorAccent">
            @color/colorAccent</item>
      <item name="android:textColor">
            #FF0000</item>
      <item name="android:textSize"> 22sp</item>
  </style>
</resources>
```

Here are a couple of notes on that:

- The `parent` attribute is important. It expresses that we want to create a theme overriding parts of the `Theme.AppCompat` theme from the compatibility library.
- Because of the naming schema Theme + DOT + AppCompat, we can infer that the theme `Theme.AppCompat` inherits from theme `Theme`. This dot-induced inheritance could have more elements.
- Instead of the parent `Theme.AppCompat`, we could use one of its subthemes. You can see a list of them; inside Android Studio click the AppCompat part and then press Ctrl+B. Android Studio will open a file with the list of all subthemes, for example `Theme.AppCompat.CompactMenu`, `Theme.AppCompat.Light`, and more.
- In the example, we see two methods to overwrite styles. Those with `android:` at the beginning refer to style settings as defined for UI elements the same way as if we want to set styles from inside a layout file. You find all of them in the online API documentation for all the views. Better, however, you use those without `android:` at the beginning because those refer to abstract style identifiers that actually make up a theme. You get a list of possible item names if inside the online documentation you search for *R.styleable.Theme*.
- The styling system has become complex over the years. If you are brave and have some time, you can navigate through all the files in Android Studio by repeatedly pressing Ctrl+B on the parents.
- `@color/...` refer to entries inside the `res/values/colors.xml` files. You should adopt that method and define new colors in your app module's `res/values/colors.xml` file.
- The values of `<item>` elements can refer to styles via `@style/...`. For example, use the item `<item name="buttonStyle">@style/Widget.AppCompat.Button</item>`. You can overwrite such items as well; just define your own styles in `styles.xml` and refer to them.

To use that new theme for your whole app at once, you write the following in the manifest file `AndroidManifest.xml`:

```
<manifest ... >
  <application android:theme="@style/MyTheme" ... >
  </application>
</manifest>
```

> **Note** You don't have to use a complete theme to overwrite styles. Instead, you can overwrite or create single styles that you can then apply to single widgets. Using a theme, however, greatly improves the design consistency of your app.

You can assign styles to different API levels. For that aim, for example, create a folder called `res/values-v21/` or any level number that suits you. Styles from inside the folder then get applied *additionally* if the current API level is *greater or equal* than that number.

Fonts in XML

Android versions starting at API level 26 (Android 8.0), and also prior versions if using the support library 26, allow you to add your own fonts in TTF or OTF format.

> **Note** To use this support library, inside your module's `build.gradle` file, add `implementation 'com.android.support:appcompat-v7:26.1.0'` in the `dependencies` section.

To add font files, create a fonts resource directory: select New ➤ Android resource directory, enter **font** as a directory name, enter **font** as a resource type, and click OK. Copy your font files to that new resource directory, but first convert all the file names to only contain characters that are allowed (*a* to *z*, 0 to 9, _) before the suffix.

To apply the new font, use the `android:fontFamily` attribute as follows:

```
<TextView ...
  android:fontFamily="@font/<FONT_NAME>"
/>
```

Here, `<FONT_NAME>` is the file name of the font without a suffix.

To add fonts with different font styles, say you have the fonts `myfont_regular.ttf`, `myfont_bold.ttf` , `myfont_italic.ttf`, and `myfont_bold_italic.ttf` inside the `font` resource folder. Add the file `myfont.xml` by choosing New ➤ Font resource file. In this file write the following:

```
<?xml version="1.0" encoding="utf-8"?>
<font-family
    xmlns:android=
         "http://schemas.android.com/apk/res/android"
    xmlns:app=
         "http://schemas.android.com/apk/res-auto">
  <font
          android:fontStyle="normal"
          app:fontStyle="normal"
          android:fontWeight="400"
          app:fontWeight="400"
          android:font="@font/myfont_regular"
          app:font="@font/myfont_regular"/>
  <font
          android:fontStyle="normal"
          app:fontStyle="normal"
          android:fontWeight="700"
          app:fontWeight="700"
          android:font="@font/myfont_bold"
          app:font="@font/myfont_bold"/>
  <font
          android:fontStyle="italic"
          app:fontStyle="italic"
          android:fontWeight="400"
          app:fontWeight="400"
          android:font="@font/myfont_italic"
          app:font="@font/myfont_italic"/>
  <font
          android:fontStyle="italic"
          app:fontStyle="italic"
          android:fontWeight="700"
          app:fontWeight="700"
          android:font="@font/myfont_bold_italic"
          app:font="@font/myfont_bold_italic"/>
</font-family>
```

Ignore Android Studio's version warning; for compatibility, all attributes use a standard and additionally a compatibility namespace.

Then you can use the name of this XML file, without the suffix, for UI views inside the `android:fontFamily` attribute.

```
<TextView ...
  android:fontFamily="@font/myfont"
  android:textStyle="normal"
/>
```

As `textStyle` you could now also use `italic` or `bold` or `bold|italic`.

2D Animation

Animation makes your apps look fancier, and while too much animation might look kinky, the right amount of it helps your users understand what your app does.

The Android OS provides several animation techniques you can use, and we describe them in the following sections.

Auto-animating Layouts

An easy way to add animation is using the built-in automatic animation for layouts. All you have to do is add `android:animateLayoutChanges="true"` to the layout declaration. Here's an example:

```
<LinearLayout
  ...
  android:animateLayoutChanges="true"
  ...
/>
```

Animated Bitmaps

You can add animation to bitmaps by providing a number of different versions of a bitmap and letting Android switch between them. First add all the images to `res/drawable`, for example `img1.png`, `img2.png`, ..., `img9.png`. Then create a file inside the same folder named, for example, `anim.xml` and write the following in the file:

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list
        xmlns:android=
          "http://schemas.android.com/apk/res/android"
        android:oneshot="false">
    <item android:drawable="@drawable/img1"
          android:duration="250" />
    <item android:drawable="@drawable/img2"
          android:duration="250" />
    ...
    <item android:drawable="@drawable/img9"
          android:duration="250" />
</animation-list>
```

Here, the duration for each bitmap slide is given in milliseconds. To make it nonrepeating, set `android:oneshot="true"`. Add the image as an `ImageView` to your layout as follows:

```
<ImageView
    android:id="@+id/img"
    android:adjustViewBounds="true"
    android:scaleType="centerCrop"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@drawable/anim1" />
```

This prepares the animation, but it needs to be started from inside the program as follows:

```
img.setBackgroundResource(R.drawable.anim1)
img.setOnClickListener{
    val anim = img.background as AnimationDrawable
    anim.start()
}
```

Here, the animation starts when the user clicks the image.

Property Animation

The property animation framework allows you to animate anything you might think of. Using the class `android.animation.ValueAnimator`, you can specify the following:

- Duration and repeating mode
- Type of the interpolated value
- Time interpolation during the animation
- A listener for value updates

Most of the time you will, however, use the `android.animation.ObjectAnimator` class because it already targets objects and their properties, so you don't have to implement listeners. This class has various static factory methods to create instances of `ObjectAnimator`. In the arguments you specify the object to animate, the name of the property of this object to use, and the values to use during the animation. On the object you can then set a certain interpolator (the default is `AccelerateDecelerateInterpolator`, which starts and ends slowly and in between accelerates and decelerates) and add a value update listener (set in `onAnimationUpdate()`) if, for example, a target object of type `View` needs to be informed that it must update itself (by calling `invalidate()`).

As an example, we define a `TextView` object inside a `FrameLayout` and move it from `x=0` to `x=500` in an accelerating manner. The layout file contains the following:

```
<FrameLayout
      android:id="@+id/fl"
      android:layout_width="match_parent"
      android:layout_height="400dp">
      <TextView
          android:id="@+id/tv"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="XXX"/>
</FrameLayout>
```

Inside the code, for example, after a button is clicked, write the following:

```
val anim = ObjectAnimator.ofFloat(tv, "x", 0.0f, 500.0f)
    .apply {
      duration = 1000 // default is 300ms
      interpolator = AccelerateInterpolator()
    }
anim.start()
```

> **Note** This works only if the object in question, here a `TextView`, has a setter method for the property specified, here a `x`. More precisely, the object needs a `setX(Float)`, which however is the case for all `View` objects.

> **Caution** Your mileage varies largely depending on the layout your UI object is placed in. After all, the layout object might have its own idea how to position objects, thwarting the animation. A `FrameLayout` is quite handsome here, but you have to do all layout inside the code.

Using `android.animation.AnimatorSet`, you can also choreograph a set of several animations. The online API documentation will tell you more about how to use it.

View Property Animator

With certain constraints imposed, a couple of `View`-related properties can also be animated using `android.view.ViewPropertyAnimator`. This seems to be less invasive compared to general property animation, but only a small number of properties can be animated, and only the drawing gets affected by the animation. The position for on-click listeners gets orphaned if views move away.

In addition, you can use it to translate views, scale views, rotate views, and fade in or fade out views. For details, please see the online documentation.

As an extension to the built-in View Property animation, you might want to take a look at the Fling animation. This type of animation applies a frictional force on moving objects, letting the animation appear more natural. To find information about the Fling animation, search for *android fling animation* inside your favorite search engine.

Spring Physics

Adding spring physics to your animations improves the user experience by making moves more realistic. To add spring physics, you need to include the corresponding support library. In your module's `build.gradle` file, add `implementation 'com.android.support:support-dynamic-animation:27.1.0'` in the `dependencies` element.

For details, refer to the online API documentation of class `android.support.animation.SpringAnimation`. The following are the most important settings:

- Inside the constructor, set the property. Available are properties for alpha, translation, rotation, scroll value, and scale.
- Add listeners; use `addUpdateListener()` and/or `addEndListener()`.
- Use `setStartVelocity()`to start the animation with an initial speed (the default is 0.0f) .

- Use `getSpring().setDampingRatio()` or in Kotlin just `.spring.dampingRatio = ...` to set the damping factor.
- Use `getSpring().setStiffness()` or in Kotlin just `.spring.stiffness = ...` to set the spring stiffness.

Use `start()` or `animateToFinalPosition()` to start the animation. This must happen inside the GUI thread or inside `runOnUiThread { ... }`.

The following is an example of a spring animation after a button press:

```
val springAnim = SpringAnimation(tv, DynamicAnimation.
 TRANSLATION_X, 500.0f).apply {
  setStartVelocity(1.0f)
  spring.stiffness =
      SpringForce.STIFFNESS_LOW
  spring.dampingRatio =
      SpringForce.DAMPING_RATIO_LOW_BOUNCY
}
springAnim.start()
```

Transitions

The transition framework allows you to apply animated transitions between different layouts. You basically create `android.transition.Scene` objects from a starting layout and an end layout and then let `TransitionManager` act. Here's an example:

```
val sceneRoot:ViewGroup = ...

// Obtain the view hierarchy to add as a child of
// the scene root when this scene is entered
val startViewHierarchy:ViewGroup = ...

// Same for the end scene
val endViewHierarchy:ViewGroup = ...

// Create the scenes
val startScene = Scene(sceneRoot, startViewHierarchy)
val endScene = Scene(sceneRoot, endViewHierarchy)

val fadeTransition = Fade()
TransitionManager.go(endScene, fadeTransition)
```

`sceneRoot` could, for instance, be a `FrameLayout` layout with the transition supposed to happen inside. You'd then at the beginning add the starting layout (`startViewHierarchy`) inside. The previous code will establish a transition to the end layout (`endViewHierarchy`), all happening *inside* the `sceneRoot`.

Such transitions can be specified from inside the code but also as special XML files. For details, please see the online documentation.

> **Caution** Certain restrictions apply. Not all `View` types will correctly take part in such layout transitions. It is, however, possible to exclude elements from the transition by using the `removeTarget()` method.

Start an Activity Using Transitions

While one activity gets replaced by another activity, it is possible to specify an exit transition for the first, specify an enter transition for the second, and allow for a smooth transition of common view elements. Such transitions can be specified either by special XML files or from inside the code. We briefly describe the latter for the XML way; for more details, please consult the online documentation.

Such switch transitions are available for API levels 21 (Android 5.0) and up. To make sure your code works with versions prior to that, inside the following code snippets we write a check:

```
if (Build.VERSION.SDK_INT >=
      Build.VERSION_CODES.LOLLIPOP) ...
```

To set an *exit* and an *enter* transition, use the following snippet inside both activities:

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

    if (Build.VERSION.SDK_INT >=
          Build.VERSION_CODES.LOLLIPOP) {
        with(window) {
            requestFeature(
                  Window.FEATURE_CONTENT_TRANSITIONS)
            exitTransition = Explode()
            // if inside the CALLED transition,
            // instead use:
            // enterTransition = Explode()

            // use this in the CALLED transition to
            // primordially start the enter transition:
            // allowEnterTransitionOverlap = true
        }
    } else {
        // Go without transition - this can be empty
    }

    ...
}
```

Here, instead of `Explode()`, you can choose `Slide()`, `Fade()`, or `AutoTransition()`. `Slide()` and `Fade()` do the obvious thing; `AutoTransition` fades out elements that are not common, moves and resizes common elements, and then fades in new elements.

> **Note** The `requestFeature()` method must happen at the beginning of `onCreate()`.

The transition gets activated only when you start a new activity via the following:

```
if (Build.VERSION.SDK_INT >=
      Build.VERSION_CODES.LOLLIPOP) {
  startActivity(intent,
      ActivityOptions.
      makeSceneTransitionAnimation(this).toBundle())
}else{
  startActivity(intent)
}
```

When the *called* activity exists, you could use `Activity.finishAfterTransition()` instead of the usual `finish()` to have the reverse transition more nicely handled. Again, you have to put that inside an `if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP)` check.

To improve user experience, you can identify elements that are common to both activities and let the transition framework handle them in a special way. To do so, you do two things.

- Give the common UI elements a special *transition name*.

  ```
  if (Build.VERSION.SDK_INT >=
        Build.VERSION_CODES.LOLLIPOP) {
    img.transitionName = "imgTrans"
        // for a certain "img" UI element
  }
  ```

  This name must be unique, and the setting should happen inside `onCreate()` in both the calling and called activities.

- Replace the `ActivityOptions.makeSceneTransitionAnimation(this)` with the following:

  ```
  ActivityOptions.
      makeSceneTransitionAnimation(
          this@MainActivity,
          UPair.create(img,"imgTrans"),
          ...more pairs...
      )
  ```

Here, `UPair` is an important alias to avoid name clashes: `import android.util.Pair as UPair`.

You can then use the `AutoTransition()` transition to get such common UI elements handled in a special way during the animation.

Fast Graphics OpenGL ES

For Android apps you can use the industry-standard OpenGL ES to render high-performance graphics in 2D and 3D. The user interface development differs significantly from the standard Android OS way, and you have to expect to spend some time learning how to use OpenGL ES. In the end, however, you might end up with outstanding graphics that pay off for the effort.

> **Note** It doesn't make sense to use OpenGL ES if the Android OS user interface provides the same functionality and performance is not an issue.

OpenGL ES comes in different versions: 1.*x*, 2.0, and 3.*x*. While there is a huge difference in methodology between 1.*x* and the later versions, the difference between 2.0 and 3.*x* is not so big. But setting 3.*x* as a strict requirement, you miss about one-third of all possible users (by beginning 2018), so the recommendation is this:

- Develop for OpenGL ES 2.0 and only if you really need it add 3.*x* features. If using 3.*x*, you best supply fallbacks for devices that do not speak 3.*x*.

In the following sections, we will be talking about version 2.0.

> **Note** OpenGL ES for Android is supported by both the framework and also the Native Development Kit (NDK). In the book we will be putting more weight on the framework classes.

OpenGL ES is extremely versatile, and usage patterns are potentially endless. Covering all that OpenGL ES provides goes beyond the scope of the book, but I will present the following scenarios to get you started:

- Configure the Activity to use OpenGL ES
- Provide a custom `GLSurfaceView` class to hold the OpenGL scene
- Provide two graphics primitives: a triangle that uses a vertex buffer and a shader program, and a quad that uses a vertex buffer, an index buffer, and a shader program
- Provide a renderer to paint the graphics
- Briefly outline how to introduce view projection
- Briefly outline how to add motion
- Briefly outline how to add light
- Briefly outline how to react to user input

Showing an OpenGL Surface in Your Activity

You can make a custom OpenGL view element be the only UI element to show in an activity via the following:

```
class MyActivity : AppCompatActivity() {
  var glView:GLSurfaceView? = null

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

      // Create a GLSurfaceView instance and set it
      // as the ContentView for this Activity.
      glView = MyGLSurfaceView(this)
      setContentView(glView)
  }
}
```

Here, `MyGLSurfaceView` is the custom `GLSurfaceView` we will define.

Or, you could use a normal XML layout file and add the custom `GL` view there by, for example, writing the following:

```
<com.example.opengl.glapp.MyGLSurfaceView
    android:layout_width="400dp"
    android:layout_height="400dp"/>
```

Here, you have to specify the full class path of the custom `GL` view class.

Creating a Custom OpenGL View Element

A custom OpenGL view element could be as easy as subclassing `android.opengl.GLSurfaceView` and specifying a renderer for the graphics data, a rendering mode, and maybe listeners for user interactions. We, however, want to go a step further and include an OpenGL ES version check so you can decide whether the inclusion of OpenGL ES 3.*x* constructs is possible. The code reads as follows:

```
import android.app.ActivityManager
import android.content.Context
import android.opengl.GLSurfaceView
import android.util.Log
import javax.microedition.khronos.egl.EGL10

class MyGLSurfaceView(context: Context) :
      GLSurfaceView(context) {
  val renderer: MyGLRenderer
  var supports3x = false
  var minVers = 0
```

```
init {
    fetchVersion()

    // Create an OpenGL ES 2.0 context
    setEGLContextClientVersion(2)

    // We set the 2.x context factory to use for
    // the view
    setEGLContextFactory()

    // We set the renderer for drawing the graphics
    renderer = MyGLRenderer()
    setRenderer(renderer)

    // This setting prevents the GLSurfaceView frame
    // from being redrawn until you call
    // requestRender()
    renderMode = GLSurfaceView.RENDERMODE_WHEN_DIRTY
}

private fun fetchVersion() {
    val activityManager =
          context.getSystemService(
              Context.ACTIVITY_SERVICE)
          as ActivityManager
    val configurationInfo =
          activityManager.deviceConfigurationInfo
    val vers = configurationInfo.glEsVersion
        // e.g. "2.0"
    supports3x = vers.split(".")[0] == "3"
    minVers = vers.split(".")[1].toInt()
    Log.i("LOG", "Supports OpenGL 3.x = " +
        supports3x)
    Log.i("LOG", "OpenGL minor version = " +
        minVers)
}

private fun setEGLContextFactory() {
    val EGL_CONTEXT_CLIENT_VERSION = 0x3098
        // from egl.h c-source
    class ContextFactory :
          GLSurfaceView.EGLContextFactory {
      override fun createContext(egl: EGL10,
            display: javax.microedition.khronos.
                     egl.EGLDisplay?,
            eglConfig: javax.microedition.khronos.
                       egl.EGLConfig?)
      :javax.microedition.khronos.egl.EGLContext? {
          val attrib_list =
              intArrayOf(EGL_CONTEXT_CLIENT_VERSION,
                         2, EGL10.EGL_NONE)
          val ectx = egl.eglCreateContext(display,
```

```
                eglConfig,
                EGL10.EGL_NO_CONTEXT,
                attrib_list)
          return ectx
        }

        override fun destroyContext(egl: EGL10,
              display: javax.microedition.khronos.
                       egl.EGLDisplay?,
              context: javax.microedition.khronos.
                       egl.EGLContext?) {
            egl.eglDestroyContext(display, context)
        }
      }
      setEGLContextFactory(ContextFactory())
  }
}
```

You can then use `.supports3x` to see whether OpenGL ES 3.*x* is supported and use `.minVers` for the minor version number if you need it. The renderer used by this class gets defined in a moment. Also, note that by virtue of `RENDERMODE_WHEN_DIRTY` a redrawing happens only on demand. If you need fully dynamic changes, then comment that line out.

A Triangle with a Vertex Buffer

A class responsible for drawing graphics primitives, in this case a simple triangle, reads as follows:

```
class Triangle {
  val vertexShaderCode = """
      attribute vec4 vPosition;
      void main() {
        gl_Position = vPosition;
      }
      """.trimIndent()

  val fragmentShaderCode = """
      precision mediump float;
      uniform vec4 vColor;
      void main() {
        gl_FragColor = vColor;
      }
      """.trimIndent()

  var program:Int? = 0

  val vertexBuffer: FloatBuffer

  var color = floatArrayOf(0.6f, 0.77f, 0.22f, 1.0f)
```

```
  var positionHandle: Int? = 0
  var colorHandle: Int? = 0

  val vertexCount =
        triangleCoords.size / COORDS_PER_VERTEX
  val vertexStride = COORDS_PER_VERTEX * 4
        // 4 bytes per vertex

  companion object {
      // number of coordinates per vertex
      internal val COORDS_PER_VERTEX = 3
      internal var triangleCoords =
            floatArrayOf( // in counterclockwise order:
              0.0f, 0.6f, 0.0f,   // top
              -0.5f, -0.3f, 0.0f, // bottom left
              0.5f, -0.3f, 0.0f   // bottom right
      )
}
```

Inside the class's `init` block, the shaders get loaded and initialized, and a vertex buffer gets prepared.

```
init {
     val vertexShader = MyGLRenderer.loadShader(
           GLES20.GL_VERTEX_SHADER,
           vertexShaderCode)
     val fragmentShader = MyGLRenderer.loadShader(
           GLES20.GL_FRAGMENT_SHADER,
           fragmentShaderCode)

     // create empty OpenGL ES Program
     program = GLES20.glCreateProgram()

     // add the vertex shader to program
     GLES20.glAttachShader(program!!, vertexShader)

     // add the fragment shader to program
     GLES20.glAttachShader(program!!, fragmentShader)

     // creates OpenGL ES program executables
     GLES20.glLinkProgram(program!!)

     // initialize vertex byte buffer for shape
     // coordinates
     val bb = ByteBuffer.allocateDirect(
              // (4 bytes per float)
              triangleCoords.size * 4)
     // use the device hardware's native byte order
     bb.order(ByteOrder.nativeOrder())

     // create a floating point buffer from bb
     vertexBuffer = bb.asFloatBuffer()
     // add the coordinates to the buffer
```

```
     vertexBuffer.put(triangleCoords)
     // set the buffer to start at 0
     vertexBuffer.position(0)
}
```

The `draw()` method performs the rendering work. Note that, as usual in OpenGL rendering, this method must run really fast. Here we just move around references:

```
fun draw() {
    // Add program to OpenGL ES environment
    GLES20.glUseProgram(program!!)

    // get handle to vertex shader's vPosition member
    positionHandle = GLES20.glGetAttribLocation(
          program!!, "vPosition")

    // Enable a handle to the triangle vertices
    GLES20.glEnableVertexAttribArray(
          positionHandle!!)

    // Prepare the triangle coordinate data
    GLES20.glVertexAttribPointer(positionHandle!!,
            COORDS_PER_VERTEX,
            GLES20.GL_FLOAT, false,
            vertexStride, vertexBuffer)

    // get handle to fragment shader's vColor member
    colorHandle = GLES20.glGetUniformLocation(
          program!!, "vColor")

    // Set color for drawing the triangle
    GLES20.glUniform4fv(colorHandle!!, 1, color, 0)

    // Draw the triangle
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0,
          vertexCount)

    // Disable vertex array
    GLES20.glDisableVertexAttribArray(
          positionHandle!!)
  }
}
```

To see how this triangle class gets used from inside the renderer, please see the following section.

A Quad with a Vertex Buffer and an Index Buffer

In OpenGL, polygons with more than three vertices best get described as gluing together as many triangles as necessary. So, for a quad we need two triangles. Obviously, some vertices then show up several times: if we have a quad A-B-C-D, we need to declare the triangles A-B-C and A-C-D, so the vertices A and C get used twice each.

Uploading vertices to the graphics hardware several times is not a good solution, and that is why there are *index lists*. We upload vertices A, B, C, and D, and in addition a list 0, 1, 3, and 2, pointing *into* the vertices list and describing the two triangles as a triangle strip (first is 0-1-3, second is 1-3-2). The corresponding code for a quad reads as follows:

```
class Quad {
  val vertexBuffer: FloatBuffer
  val drawListBuffer: ShortBuffer

  val vertexShaderCode = """
      attribute vec4 vPosition;
      void main() {
        gl_Position = vPosition;
      }
      """.trimIndent()

  val fragmentShaderCode = """
      precision mediump float;
      uniform vec4 vColor;
      void main() {
        gl_FragColor = vColor;
      }
      """.trimIndent()

  // The shader program
  var program:Int? = 0

  var color = floatArrayOf(0.94f, 0.67f, 0.22f, 1.0f)

  val vbo = IntArray(1) // one vertex buffer
  val ibo = IntArray(1) // one index buffer

    var positionHandle: Int? = 0
    var colorHandle: Int? = 0

    companion object {
        val BYTES_PER_FLOAT = 4
        val BYTES_PER_SHORT = 2
        val COORDS_PER_VERTEX = 3
        val VERTEX_STRIDE = COORDS_PER_VERTEX *
              BYTES_PER_FLOAT
        var quadCoords = floatArrayOf(
                -0.5f, 0.2f, 0.0f, // top left
                -0.5f, -0.5f, 0.0f, // bottom left
                0.2f, -0.5f, 0.0f, // bottom right
                0.2f, 0.2f, 0.0f) // top right
        val drawOrder = shortArrayOf(0, 1, 3, 2)
              // order to draw vertices
}
```

As for the previous triangle, we initialize the shaders and the buffers in the `init` block.

```
init {
    val vertexShader = MyGLRenderer.loadShader(
          GLES20.GL_VERTEX_SHADER,
          vertexShaderCode)
  val fragmentShader = MyGLRenderer.loadShader(
          GLES20.GL_FRAGMENT_SHADER,
          fragmentShaderCode)

  program = GLES20.glCreateProgram().apply {
      // add the vertex shader to program
      GLES20.glAttachShader(this, vertexShader)

      // add the fragment shader to program
      GLES20.glAttachShader(this, fragmentShader)

      // creates OpenGL ES program executables
      GLES20.glLinkProgram(this)
  }

  // initialize vertex byte buffer for shape coords
  vertexBuffer = ByteBuffer.allocateDirect(
          quadCoords.size * BYTES_PER_FLOAT).apply{
      order(ByteOrder.nativeOrder())
  }.asFloatBuffer().apply {
      put(quadCoords)
      position(0)
  }

  // initialize byte buffer for the draw list
  drawListBuffer = ByteBuffer.allocateDirect(
          drawOrder.size * BYTES_PER_SHORT).apply {
      order(ByteOrder.nativeOrder())
  }.asShortBuffer().apply {
      put(drawOrder)
      position(0)
  }

  GLES20.glGenBuffers(1, vbo, 0);
  GLES20.glGenBuffers(1, ibo, 0);
  if (vbo[0] > 0 && ibo[0] > 0) {
      GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER,
          vbo[0])
      GLES20.glBufferData(GLES20.GL_ARRAY_BUFFER,
          vertexBuffer.capacity() * BYTES_PER_FLOAT,
          vertexBuffer, GLES20.GL_STATIC_DRAW)
      GLES20.glBindBuffer(
          GLES20.GL_ELEMENT_ARRAY_BUFFER, ibo[0])
      GLES20.glBufferData(
          GLES20.GL_ELEMENT_ARRAY_BUFFER,
```

```
            drawListBuffer.capacity() *
                  BYTES_PER_SHORT,
            drawListBuffer, GLES20.GL_STATIC_DRAW)

        GLES20.glBindBuffer(
            GLES20.GL_ARRAY_BUFFER, 0);
        GLES20.glBindBuffer(
            GLES20.GL_ELEMENT_ARRAY_BUFFER, 0)
    } else {
        //TODO: some error handling
    }
}
```

The `draw()` method performs the rendering, as is the case for the triangle class we described earlier.

```
fun draw() {
    // Add program to OpenGL ES environment
    GLES20.glUseProgram(program!!)

    // Get handle to fragment shader's vColor member
    colorHandle = GLES20.glGetUniformLocation(
          program!!, "vColor")
    // Set color for drawing the quad
    GLES20.glUniform4fv(colorHandle!!, 1, color, 0)

    // Get handle to vertex shader's vPosition member
    positionHandle = GLES20.glGetAttribLocation(
          program!!, "vPosition")

    // Enable a handle to the vertices
    GLES20.glEnableVertexAttribArray(
          positionHandle!!)

    // Prepare the coordinate data
    GLES20.glVertexAttribPointer(positionHandle!!,
          COORDS_PER_VERTEX,
          GLES20.GL_FLOAT, false,
          VERTEX_STRIDE, vertexBuffer)

    // Draw the quad
    GLES20.glBindBuffer(
          GLES20.GL_ARRAY_BUFFER, vbo[0]);

    // Bind Attributes
    GLES20.glBindBuffer(
          GLES20.GL_ELEMENT_ARRAY_BUFFER, ibo[0])
    GLES20.glDrawElements(GLES20.GL_TRIANGLE_STRIP,
          drawListBuffer.capacity(),
          GLES20.GL_UNSIGNED_SHORT, 0)
```

```
    GLES20.glBindBuffer(
          GLES20.GL_ARRAY_BUFFER, 0)
    GLES20.glBindBuffer(
          GLES20.GL_ELEMENT_ARRAY_BUFFER, 0)

    // Disable vertex array
    GLES20.glDisableVertexAttribArray(
          positionHandle!!)
  }
}
```

In the constructor of the shader program, the vertex buffer and the index buffer get uploaded to the graphics hardware. Inside the `draw()` method, which potentially gets called often, only pointers to the uploaded buffers get used. The usage of this `Quad` class gets described in the following section.

Creating and Using a Renderer

A renderer is responsible for drawing the graphics objects. Since we are using a subclass of `android.opengl.GLSurfaceView`, the renderer must be a subclass of `GLSurfaceView.Renderer`. Since the classes `Triangle` and `Quad` have their own shaders, all the renderer needs to do is instantiate a quad and a triangle and use their `draw()` methods, besides some boilerplate code.

```
class MyGLRenderer : GLSurfaceView.Renderer {
  companion object {
      fun loadShader(type: Int, shaderCode: String)
            : Int {
        // create a vertex shader type
        //      (GLES20.GL_VERTEX_SHADER)
        // or a fragment shader type
        //      (GLES20.GL_FRAGMENT_SHADER)
        val shader = GLES20.glCreateShader(type)

        // add the source code to the shader and
        // compile it
        GLES20.glShaderSource(shader, shaderCode)
        GLES20.glCompileShader(shader)

        return shader
      }
  }

  var triangle:Triangle? = null
  var quad:Quad? = null

  // Called once to set up the view's OpenGL ES
  // environment.
  override
```

```
  fun onSurfaceCreated(gl: GL10?, config:
        javax.microedition.khronos.egl.EGLConfig?) {
    // enable face culling feature
    GLES20.glEnable(GL10.GL_CULL_FACE)
    // specify which faces to not draw
    GLES20.glCullFace(GL10.GL_BACK)
    // Set the background frame color
    GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f)
  }

  // Called for each redraw of the view.
  // If renderMode =
  // GLSurfaceView.RENDERMODE_WHEN_DIRTY
  // (see MyGLSurfaceView)
  // this will not be called every frame
  override
  fun onDrawFrame(unused: GL10) {
    // Redraw background color
    GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT)

    triangle = triangle ?: Triangle()
    triangle?.draw()
    quad = quad ?: Quad()
    quad?.draw()
  }

  override
  fun onSurfaceChanged(unused: GL10, width: Int,
        height: Int) {
    GLES20.glViewport(0, 0, width, height)
  }
}
```

Projection

Once we start using the third dimension, we need to talk about projection. The projection describes how the three dimensions of the vertices get mapped to two-dimensional screen coordinates. The `Triangle` and `Quad` graphics primitives we built so far both use their own shader program. While this gives us the maximum flexibility, to avoid a proliferation of shader programs, it is better to extract the shader program and use just one from inside the renderer. Also, the projection calculation then needs to be done at only one place.

In addition, we let the renderer prepare `N` times two buffer objects, one pair of vertex and index buffer per object, and provide corresponding handles to each graphics primitive constructor. The new `Square` class then looks like the following:

```
class Square(val program: Int?,
             val vertBuf:Int, val idxBuf:Int) {
  val vertexBuffer: FloatBuffer
  val drawListBuffer: ShortBuffer
```

```
  var color = floatArrayOf(0.94f, 0.67f, 0.22f, 1.0f)

  companion object {
      val BYTES_PER_FLOAT = 4
      val BYTES_PER_SHORT = 2
      val COORDS_PER_VERTEX = 3
      val VERTEX_STRIDE = COORDS_PER_VERTEX *
          BYTES_PER_FLOAT
      var coords = floatArrayOf(
              -0.5f, 0.2f, 0.0f, // top left
              -0.5f, -0.5f, 0.0f, // bottom left
              0.2f, -0.5f, 0.0f, // bottom right
              0.2f, 0.2f, 0.0f) // top right
      val drawOrder = shortArrayOf(0, 1, 3, 2)
            // order to draw vertices
}
```

The class no longer contains shader code, so what is left for the `init` block is preparing the buffers to use the following:

```
init {
    // initialize vertex byte buffer for shape
    // coordinates
    vertexBuffer = ByteBuffer.allocateDirect(
            coords.size * BYTES_PER_FLOAT).apply{
        order(ByteOrder.nativeOrder())
    }.asFloatBuffer().apply {
        put(coords)
        position(0)
    }

    // initialize byte buffer for the draw list
    drawListBuffer = ByteBuffer.allocateDirect(
            drawOrder.size * BYTES_PER_SHORT).apply {
        order(ByteOrder.nativeOrder())
    }.asShortBuffer().apply {
        put(drawOrder)
        position(0)
    }

    if (vertBuf > 0 && idxBuf > 0) {
        GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER,
              vertBuf)
        GLES20.glBufferData(GLES20.GL_ARRAY_BUFFER,
                vertexBuffer.capacity() *
                      BYTES_PER_FLOAT,
                vertexBuffer, GLES20.GL_STATIC_DRAW)

        GLES20.glBindBuffer(
              GLES20.GL_ELEMENT_ARRAY_BUFFER, idxBuf)
        GLES20.glBufferData(
              GLES20.GL_ELEMENT_ARRAY_BUFFER,
```

```
                drawListBuffer.capacity() *
                        BYTES_PER_SHORT,
                drawListBuffer, GLES20.GL_STATIC_DRAW)

            GLES20.glBindBuffer(
                    GLES20.GL_ARRAY_BUFFER, 0)
            GLES20.glBindBuffer(
                    GLES20.GL_ELEMENT_ARRAY_BUFFER, 0)
        } else {
            //TODO: error handling
        }
}
```

The `draw()` method does not substantially differ from before. This time we use the shader program provided in the constructor. Again, this method runs fast, since it only shifts around references.

```
fun draw() {
    // Add program to OpenGL ES environment
    GLES20.glUseProgram(program!!)

    // get handle to fragment shader's vColor member
    val colorHandle = GLES20.glGetUniformLocation(
          program!!, "vColor")
    // Set color for drawing the square
    GLES20.glUniform4fv(colorHandle!!, 1, color, 0)

    // get handle to vertex shader's vPosition member
    val positionHandle = GLES20.glGetAttribLocation(
          program!!, "vPosition")

    // Enable a handle to the vertices
    GLES20.glEnableVertexAttribArray(
          positionHandle!!)

    // Prepare the coordinate data
    GLES20.glVertexAttribPointer(positionHandle!!,
            COORDS_PER_VERTEX,
            GLES20.GL_FLOAT, false,
            VERTEX_STRIDE, vertexBuffer)

    // Draw the square
    GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER,
          vertBuf)
    GLES20.glBindBuffer(
          GLES20.GL_ELEMENT_ARRAY_BUFFER, idxBuf)
    GLES20.glDrawElements(GLES20.GL_TRIANGLE_STRIP,
            drawListBuffer.capacity(),
            GLES20.GL_UNSIGNED_SHORT, 0)
```

```
    GLES20.glBindBuffer(
          GLES20.GL_ARRAY_BUFFER, 0)
    GLES20.glBindBuffer(
          GLES20.GL_ELEMENT_ARRAY_BUFFER, 0)

    // Disable vertex array
    GLES20.glDisableVertexAttribArray(
          positionHandle!!)
  }
}
```

Here, inside the constructor we fetch handles to the program, a vertex buffer name (an integer) and an index buffer name (another integer), and furthermore we prepare and upload the vertex and index buffers to the graphics hardware. Letting a new `Triangle` class use the same method is left as an exercise for you.

The new renderer class now contains the shader program and prepares handles for the buffers. But we go one step further and also add projection matrices.

```
class MyGLRenderer : GLSurfaceView.Renderer {
  companion object {
      fun loadShader(type: Int, shaderCode: String)
            : Int {
        // create a vertex shader type
        // (GLES20.GL_VERTEX_SHADER)
        // or a fragment shader type
        // (GLES20.GL_FRAGMENT_SHADER)
        val shader = GLES20.glCreateShader(type)

        // add the source code to the shader and
        // compile it
        GLES20.glShaderSource(shader, shaderCode)
        GLES20.glCompileShader(shader)

        return shader
      }
  }

  val vertexShaderCode = """
      attribute vec4 vPosition;
      uniform mat4 uMVPMatrix;
      void main() {
        gl_Position = uMVPMatrix * vPosition;
      }
      """.trimIndent()

  val fragmentShaderCode = """
      precision mediump float;
      uniform vec4 vColor;
      void main() {
        gl_FragColor = vColor;
      }
      """.trimIndent()
```

```
  var triangle:Triangle? = null
  var square:Square? = null
  var program:Int? = 0

  val vbo = IntArray(2) // vertex buffers
  val ibo = IntArray(2) // index buffers

  val vMatrix:FloatArray = FloatArray(16)
  val projMatrix:FloatArray = FloatArray(16)
  val mvpMatrix:FloatArray = FloatArray(16)
```

The method `onSurfaceCreated()` just gets called once by the system when the OpenGL rendering is ready to go. We use it to set some rendering flags and to initialize the shaders.

```
// Called once to set up the view's
// OpenGL ES environment.
override fun onSurfaceCreated(gl: GL10?, config:
      javax.microedition.khronos.egl.EGLConfig?) {
  // enable face culling feature
  GLES20.glEnable(GL10.GL_CULL_FACE)
  // specify which faces to not draw
  GLES20.glCullFace(GL10.GL_BACK)

  // Set the background frame color
  GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f)

  val vertexShader = loadShader(
            GLES20.GL_VERTEX_SHADER,
            vertexShaderCode)
  val fragmentShader = loadShader(
            GLES20.GL_FRAGMENT_SHADER,
            fragmentShaderCode)

  // create empty OpenGL ES Program
  program = GLES20.glCreateProgram()

  // add the vertex shader to program
  GLES20.glAttachShader(program!!, vertexShader)

  // add the fragment shader to program
  GLES20.glAttachShader(program!!, fragmentShader)

  // creates OpenGL ES program executables
  GLES20.glLinkProgram(program!!)

  GLES20.glGenBuffers(2, vbo, 0) // just buffer names
  GLES20.glGenBuffers(2, ibo, 0)
```

```
  // Create a camera view and an orthogonal projection
  // matrix
  Matrix.setLookAtM(vMatrix, 0, 0f, 0f, 3.0f, 0f,
        0f, 0f, 0f, 1.0f, 0.0f)
  Matrix.orthoM(projMatrix,0,-1.0f,1.0f, -1.0f, 1.0f,
        100.0f, -100.0f)
}
```

The callback method `onDrawFrame()` gets called for each redraw of the view. If `renderMode = GLSurfaceView.RENDERMODE_WHEN_DIRTY`, see class `MyGLSurfaceView`. This will, however, not be called every frame, only if changes are detected. The following snippet also closes the class:

```
override fun onDrawFrame(unused: GL10) {
    // Redraw background color
    GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT)

    GLES20.glUseProgram(program!!)
    val muMVPMatrixHandle = GLES20.glGetUniformLocation
(
          program!!, "uMVPMatrix");
    Matrix.multiplyMM(mvpMatrix, 0,
          projMatrix, 0, vMatrix, 0)

    // Apply the combined projection and camera view
    // transformations
    GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1,
          false, mvpMatrix, 0);

    triangle = triangle ?:
          Triangle(program,vbo[0],ibo[0])
    triangle?.draw()

    square = square ?:
          Square(program,vbo[1],ibo[1])
    square?.draw()
  }

  override
  fun onSurfaceChanged(unused: GL10, width: Int,
        height: Int) {
      GLES20.glViewport(0, 0, width, height)
  }
}
```

You can see the projection matrices get calculated inside the Kotlin code, but they're uploaded as shader *uniform* variables and used from inside the vertex shader.

Applying projection to three-dimensional objects makes more sense. As a general assumption for such 3D objects, we expect the following:

- Vertex coordinates in four-dimensional homogeneous coordinates
- RGBA colors assigned to vertices
- Face normals assigned to vertices

Using four coordinate values instead of the usual three (x, y, z) helps for perspective projection. Colors assigned to vertices can be used from inside the shader code to apply a coloring scheme. But it can also be ignored or used for noncoloring purposes. This is totally up to the shader code. The normals help for a realistic lighting.

The renderer gets just new shader code, as shown here:

```
val vertexShaderCode = """
      attribute vec4 vPosition;
      attribute vec4 vNorm;
      attribute vec4 vColor;

      varying vec4 fColor;
      varying vec4 fNorm;

      uniform mat4 uMVPMatrix;

      void main() {
        gl_Position = uMVPMatrix * vPosition;
        fColor = vColor;
        fNorm = vNorm;
      }
      """.trimIndent()

  val fragmentShaderCode = """
      precision mediump float;
      varying vec4 fColor;
      varying vec4 fNorm;

      void main() {
        gl_FragColor = fColor;
      }
      """.trimIndent()
```

As a sample 3D object, I present a cube with an interpolated coloring according to the vertex colors and the normals ignored for now.

```
class Cube(val program: Int?, val vertBuf:Int,
      val idxBuf:Int) {
  val vertexBuffer: FloatBuffer
  val drawListBuffer: ShortBuffer
```

The companion object holds all the coordinates and indices we need for the cube.

```
companion object {
    val BYTES_PER_FLOAT = 4
    val BYTES_PER_SHORT = 2
    val COORDS_PER_VERTEX = 4
    val NORMS_PER_VERTEX = 4
    val COLORS_PER_VERTEX = 4
    val VERTEX_STRIDE = (COORDS_PER_VERTEX +
          NORMS_PER_VERTEX +
          COLORS_PER_VERTEX) * BYTES_PER_FLOAT
    var coords = floatArrayOf(
            // positions + normals + colors
            // --- front
            -0.2f, -0.2f, 0.2f, 1.0f,
                  0.0f, 0.0f, 1.0f, 0.0f,
                        1.0f, 0.0f, 0.0f, 1.0f,
            0.2f, -0.2f, 0.2f, 1.0f,
                  0.0f, 0.0f, 1.0f, 0.0f,
                        1.0f, 0.0f, 0.0f, 1.0f,
            0.2f, 0.2f, 0.2f, 1.0f,
                  0.0f, 0.0f, 1.0f, 0.0f,
                        1.0f, 0.0f, 0.0f, 1.0f,
            -0.2f, 0.2f, 0.2f, 1.0f,
                   0.0f, 0.0f, 1.0f, 0.0f,
                         1.0f, 0.0f, 0.0f, 1.0f,
            // --- back
            -0.2f, -0.2f, -0.2f, 1.0f,
                   0.0f, 0.0f, -1.0f, 0.0f,
                         0.0f, 1.0f, 0.0f, 1.0f,
            0.2f, -0.2f, -0.2f, 1.0f,
                  0.0f, 0.0f, -1.0f, 0.0f,
                        0.0f, 1.0f, 0.0f, 1.0f,
            0.2f, 0.2f, -0.2f, 1.0f,
                  0.0f, 0.0f, -1.0f, 0.0f,
                        0.0f, 1.0f, 0.0f, 1.0f,
            -0.2f, 0.2f, -0.2f, 1.0f,
                   0.0f, 0.0f, -1.0f, 0.0f,
                         0.0f, 1.0f, 0.0f, 1.0f,
            // --- bottom
            -0.2f, -0.2f, 0.2f, 1.0f,
                   0.0f, -1.0f, 0.0f, 0.0f,
                         0.0f, 0.0f, 1.0f, 1.0f,
            0.2f, -0.2f, 0.2f, 1.0f,
                  0.0f, -1.0f, 0.0f, 0.0f,
                        0.0f, 0.0f, 1.0f, 1.0f,
            0.2f, -0.2f, -0.2f, 1.0f,
                  0.0f, -1.0f, 0.0f, 0.0f,
                        0.0f, 0.0f, 1.0f, 1.0f,
            -0.2f, -0.2f, -0.2f, 1.0f,
                   0.0f, -1.0f, 0.0f, 0.0f,
                         0.0f, 0.0f, 1.0f, 1.0f,
            // --- top
```

```
            -0.2f, 0.2f, 0.2f, 1.0f,
                   0.0f, 1.0f, 0.0f, 0.0f,
                         1.0f, 0.0f, 1.0f, 1.0f,
            0.2f, 0.2f, 0.2f, 1.0f,
                  0.0f, 1.0f, 0.0f, 0.0f,
                        1.0f, 0.0f, 1.0f, 1.0f,
            0.2f, 0.2f, -0.2f, 1.0f,
                  0.0f, 1.0f, 0.0f, 0.0f,
                        1.0f, 0.0f, 1.0f, 1.0f,
            -0.2f, 0.2f, -0.2f, 1.0f,
                   0.0f, 1.0f, 0.0f, 0.0f,
                         1.0f, 0.0f, 1.0f, 1.0f,
            // --- right
            0.2f, -0.2f, 0.2f, 1.0f,
                  1.0f, 0.0f, 0.0f, 0.0f,
                        0.0f, 1.0f, 1.0f, 1.0f,
            0.2f, 0.2f, 0.2f, 1.0f,
                  1.0f, 0.0f, 0.0f, 0.0f,
                        0.0f, 1.0f, 1.0f, 1.0f,
            0.2f, 0.2f, -0.2f, 1.0f,
                  1.0f, 0.0f, 0.0f, 0.0f,
                        0.0f, 1.0f, 1.0f, 1.0f,
            0.2f, -0.2f, -0.2f, 1.0f,
                  1.0f, 0.0f, 0.0f, 0.0f,
                        0.0f, 1.0f, 1.0f, 1.0f,
            // --- left
            -0.2f, -0.2f, 0.2f, 1.0f,
                   -1.0f, 0.0f, 0.0f, 0.0f,
                          1.0f, 1.0f, 0.0f, 1.0f,
            -0.2f, 0.2f, 0.2f, 1.0f,
                   -1.0f, 0.0f, 0.0f, 0.0f,
                          1.0f, 1.0f, 0.0f, 1.0f,
            -0.2f, 0.2f, -0.2f, 1.0f,
                   -1.0f, 0.0f, 0.0f, 0.0f,
                          1.0f, 1.0f, 0.0f, 1.0f,
            -0.2f, -0.2f, -0.2f, 1.0f,
                   -1.0f, 0.0f, 0.0f, 0.0f,
                          1.0f, 1.0f, 0.0f, 1.0f
    )
    val drawOrder = shortArrayOf( // vertices order
            0, 1, 2,     0, 2, 3,     // front
            4, 6, 5,     4, 7, 6,     // back
            8, 10, 9,    8, 11, 10,   // bottom
            12, 13, 14   12, 14, 15,  // top
            16, 18, 17   16, 19, 18,  // right
            20, 21, 22   20, 22, 23,  // left
        )
}
```

As in the previous listings, we use the `init` block to prepare and initialize the buffers we need for the shaders.

```
init {
    // initialize vertex byte buffer for shape
    // coordinates, normals and colors
    vertexBuffer = ByteBuffer.allocateDirect(
            coords.size * BYTES_PER_FLOAT).apply{
        order(ByteOrder.nativeOrder())
    }.asFloatBuffer().apply {
        put(coords)
        position(0)
    }

    // initialize byte buffer for the draw list
    drawListBuffer = ByteBuffer.allocateDirect(
            drawOrder.size * BYTES_PER_SHORT).apply {
        order(ByteOrder.nativeOrder())
    }.asShortBuffer().apply {
        put(drawOrder)
        position(0)
    }

    if (vertBuf > 0 && idxBuf > 0) {
        GLES20.glBindBuffer(
              GLES20.GL_ARRAY_BUFFER, vertBuf)
        GLES20.glBufferData(GLES20.GL_ARRAY_BUFFER,
                vertexBuffer.capacity() *
                      BYTES_PER_FLOAT,
                vertexBuffer, GLES20.GL_STATIC_DRAW)

        GLES20.glBindBuffer(
              GLES20.GL_ELEMENT_ARRAY_BUFFER, idxBuf)
        GLES20.glBufferData(
                GLES20.GL_ELEMENT_ARRAY_BUFFER,
                drawListBuffer.capacity() *
                        BYTES_PER_SHORT,
                drawListBuffer, GLES20.GL_STATIC_DRAW)

          GLES20.glBindBuffer(
                GLES20.GL_ARRAY_BUFFER, 0)
          GLES20.glBindBuffer(
                  GLES20.GL_ELEMENT_ARRAY_BUFFER, 0)
      } else {
          // TODO: error handling
      }
}
```

The draw() method for rendering the graphics this time reads as follows:

```
fun draw() {
    // Add program to OpenGL ES environment
    GLES20.glUseProgram(program!!)

    // get handle to vertex shader's vPosition member
    val positionHandle =
        GLES20.glGetAttribLocation(program,
                                   "vPosition")
    // Enable a handle to the vertices
    GLES20.glEnableVertexAttribArray(positionHandle)
    // Prepare the coordinate data
    GLES20.glVertexAttribPointer(
            positionHandle, COORDS_PER_VERTEX,
            GLES20.GL_FLOAT, false,
            VERTEX_STRIDE, vertexBuffer)

    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    // Buffer offsets are a little bit strange in the
    // Java binding - for the normals and colors we
    // create new views and then reset the vertex
    // array
    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    // get handle to vertex shader's vPosition member
    vertexBuffer.position(COORDS_PER_VERTEX)
    val normBuffer = vertexBuffer.slice()
          // create a new view
    vertexBuffer.rewind()
          // ... and rewind the original buffer
    val normHandle = GLES20.glGetAttribLocation(
          program, "vNorm")
    if(normHandle >= 0) {
      // Enable a handle to the vertices
      GLES20.glEnableVertexAttribArray(normHandle)
      // Prepare the coordinate data
      GLES20.glVertexAttribPointer(normHandle,
          COORDS_PER_VERTEX,
          GLES20.GL_FLOAT, false,
          VERTEX_STRIDE, normBuffer)
    }

    // get handle to vertex shader's vColor member
    vertexBuffer.position(COORDS_PER_VERTEX +
                          NORMS_PER_VERTEX)
    val colorBuffer = vertexBuffer.slice()
          // create a new view
    vertexBuffer.rewind()
          // ... and rewind the original buffer
    val colorHandle = GLES20.glGetAttribLocation(
          program, "vColor")
    if(colorHandle >= 0) {
```

```
      // Enable a handle to the vertices
      GLES20.glEnableVertexAttribArray(colorHandle)
      // Prepare the coordinate data
      GLES20.glVertexAttribPointer(colorHandle,
          COLORS_PER_VERTEX,
          GLES20.GL_FLOAT, false,
          VERTEX_STRIDE, colorBuffer)
    }

    // Draw the cube
    GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER,
          vertBuf)
    GLES20.glBindBuffer(
          GLES20.GL_ELEMENT_ARRAY_BUFFER, idxBuf)
    GLES20.glDrawElements(GLES20.GL_TRIANGLES,
            drawListBuffer.capacity(),
            GLES20.GL_UNSIGNED_SHORT, 0)

    GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER,0)
GLES20.glBindBuffer(GLES20.GL_ELEMENT_ARRAY_BUFFER,0)
    // Disable attribute arrays
    GLES20.glDisableVertexAttribArray(positionHandle)
    if(normHandle >= 0)
       GLES20.glDisableVertexAttribArray(normHandle)
    if(colorHandle >= 0)
       GLES20.glDisableVertexAttribArray(colorHandle)
  }
}
```

Motion

Up to now our objects were kind of static, and by virtue of `renderMode = GLSurfaceView.RENDERMODE_WHEN_DIRTY` in class `MyGLSurfaceView`, redrawing happens only on demand. If instead you use `GLSurfaceView.RENDERMODE_CONTINUOUSLY`, the redrawing happens every frame.

Note that you still can and should use vertex and index buffers for feeding the shaders. You can easily introduce motion by adjusting the matrices inside Kotlin or by directly editing the shader code, for example after adding more *uniform* variables.

Light

Lighting can be added inside the *fragment shader* code. This time we will have to use the normal vectors because they determine how light gets reflected on the surface elements. If we introduce light, we need to tell where its position is. For this aim, inside the renderer's companion object add the following:

```
val lightPos = floatArrayOf(0.0f, 0.0f, 4.0f, 0.0f)
```

With the shader code now getting more complex, we should have to find out how to get hold of the error messages. For this to be achieved, you can add the following in the renderer's `loadShader()` function:

```
val statusShader = IntArray(1)
GLES20.glGetShaderiv(shader, GLES20.GL_COMPILE_STATUS,
      IntBuffer.wrap(statusShader))
if (statusShader[0] == GLES20.GL_FALSE) {
  val s = GLES20.glGetShaderInfoLog(shader)
  Log.e("LOG", "Shader compilation: " + s)
}
```

Similarly, after the program linking, add the following snippet:

```
val statusShader = IntArray(1)
GLES20.glGetShaderiv(program!!, GLES20.GL_LINK_STATUS,
      IntBuffer.wrap(statusShader))
if (statusShader[0] == GLES20.GL_FALSE) {
  val s = GLES20.glGetShaderInfoLog(program!!)
  Log.e("LOG", "Shader linking: " + s)
}
```

The new vertex shader transports a transformed normal vector according to rotation and scaling, *not* including translation (that is why the fourth component of the normal vectors reads 0.0), and also a transformed position vector, this time including translation.

```
val vertexShaderCode = """
    attribute vec4 vPosition;
    attribute vec4 vNorm;
    attribute vec4 vColor;

    varying vec4 fColor;
    varying vec3 N;
    varying vec3 v;

    uniform mat4 uVMatrix;
    uniform mat4 uMVPMatrix;

    void main() {
      gl_Position = uMVPMatrix * vPosition;
      fColor = vColor;
      v = vec3(uVMatrix * vPosition);
      N = normalize(vec3(uVMatrix * vNorm));
    }
    """.trimIndent()
```

Note that we also transport the vertex colors, although we are not going to use them any longer. You could, however, merge the color information, so we don't remove it. The following code ignores the vertex colors.

The new fragment shader takes the interpolated positions and normals from the vertex shader, adds a uniform variable for the light's position, and in our case uses the Phong shading model to apply light.

```
val fragmentShaderCode = """
precision mediump float;
varying vec4 fColor;
varying vec3 N;
varying vec3 v;
uniform vec4 lightPos;

void main() {
  vec3 L = normalize(lightPos.xyz - v);
  vec3 E = normalize(-v); // eye coordinates!
  vec3 R = normalize(-reflect(L,N));

  //calculate Ambient Term:
  vec4 Iamb = vec4(0.0, 0.1, 0.1, 1.0);

  //calculate Diffuse Term:
  vec4 Idiff = vec4(0.0, 0.0, 1.0, 1.0) *
      max(dot(N,L), 0.0);
  Idiff = clamp(Idiff, 0.0, 1.0);

  // calculate Specular Term:
  vec4 Ispec = vec4(1.0, 1.0, 0.5, 1.0) *
      pow(max(dot(R,E),0.0),
        /*shininess=*/5.0);
  Ispec = clamp(Ispec, 0.0, 1.0);

  // write Total Color:
  gl_FragColor = Iamb + Idiff + Ispec;
  //gl_FragColor = fColor; // use vertex color instead
}
""".trimIndent()
```

Don't forget to add a handle for the light's position inside the renderer's `onDrawFrame()` function.

```
// The light position
val lightPosHandle = GLES20.glGetUniformLocation(
      program!!, "lightPos");
GLES20.glUniform4f(lightPosHandle,
      lightPos[0],lightPos[1],lightPos[2],lightPos[3])
```

The vectors used in the Phong shading algorithm are depicted in Figure 9-2. You will find the same vector names used in the shader code.

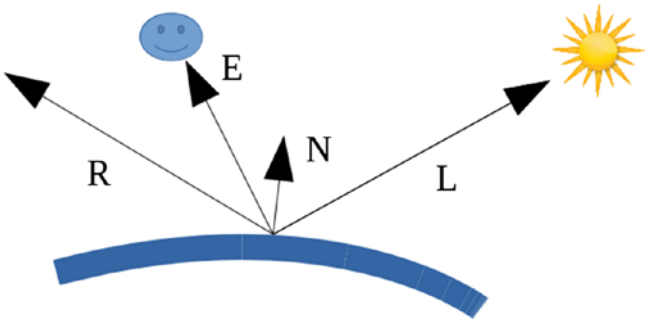

Figure 9-2. Phong shading vectors

You can add more dynamics to the lighting by using uniforms for the color components. For simplicity, I hard-coded them inside the fragment shaders (all those `vec4(...)` constructors). A lighted cube looks like Figure 9-3.

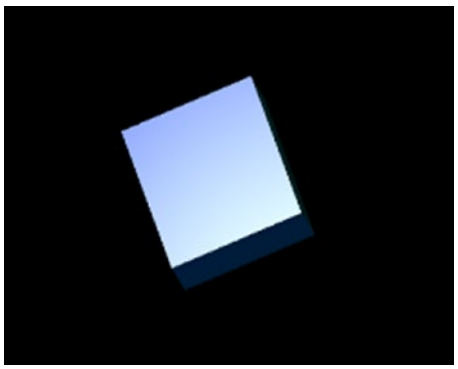

Figure 9-3. A lighted cube

Textures

Images in OpenGL get handled by *textures*, which are bitmap data uploaded to the graphics hardware and usually spanning surfaces defined by textures. To allow for textures, the companion object from the renderer we introduced in the previous sections gets another function to load texturable images from the Android resources folder.

```
companion object {
    ...
    fun loadTexture(context: Context, resourceId: Int):
          Int {
    val textureHandle = IntArray(1)

    GLES20.glGenTextures(1, textureHandle, 0)
```

```
    if (textureHandle[0] != 0) {
       val options = BitmapFactory.Options().apply {
           inScaled = false  // No pre-scaling
       }

       // Read in the resource
       val bitmap = BitmapFactory.decodeResource(
           context.getResources(),
           resourceId, options)

       // Bind to the texture in OpenGL
       GLES20.glBindTexture(GLES20.GL_TEXTURE_2D,
           textureHandle[0])

       // Set filtering
       GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
           GLES20.GL_TEXTURE_MIN_FILTER,
           GLES20.GL_NEAREST)
       GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
           GLES20.GL_TEXTURE_MAG_FILTER,
           GLES20.GL_NEAREST)

       // Load the bitmap into the bound texture.
       GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0,
           bitmap, 0)

       // The bitmap is no longer needed.
       bitmap.recycle()
   }else{
       // TODO: handle error
   }
   return textureHandle[0]
}
```

In addition, the renderer gets new code for both the vertex and the fragment shader.

```
val vertexShaderCode = """
    attribute vec4 vPosition;
    attribute vec2 vTexture;
    attribute vec4 vColor;

    varying vec2 textureCoords;
    varying vec4 fColor;

    uniform mat4 uMVPMatrix;

    void main() {
      gl_Position = uMVPMatrix * vPosition;
      textureCoords = vTexture;
      fColor = vColor;
    }
    """.trimIndent()
```

```
val fragmentShaderCode = """
    precision mediump float;
    uniform sampler2D texture;  // The input texture.
    varying vec2 textureCoords;
    varying vec4 fColor;

    void main() {
      gl_FragColor = texture2D(texture, textureCoords);
      // use vertex color instead:
      // gl_FragColor = fColor;
    }
    """.trimIndent()
```

The attribute `vTexture` corresponds to a new data section in the vertex definition of an object. `uniform sampler2D texture;` describes the connection to a texture object defined in the Kotlin code.

As a sample object, we define a plane that resembles one of the faces from the `Cube` class we defined earlier, apart from additionally feeding the texture.

```
class Plane(val program: Int?, val vertBuf:Int,
      val idxBuf:Int, val context: Context) {

   val vertexBuffer: FloatBuffer
   val drawListBuffer: ShortBuffer

   // Used to pass in the texture.
   var textureUniformHandle: Int = 0
   // A handle to our texture data
   var textureDataHandle: Int = 0
```

The companion object is used to define the coordinates and some constants.

```
companion object {
    val BYTES_PER_FLOAT = 4
    val BYTES_PER_SHORT = 2
    val COORDS_PER_VERTEX = 4
    val TEXTURE_PER_VERTEX = 2
    val NORMS_PER_VERTEX = 4
    val COLORS_PER_VERTEX = 4
    val VERTEX_STRIDE = (COORDS_PER_VERTEX +
          TEXTURE_PER_VERTEX +
          NORMS_PER_VERTEX +
          COLORS_PER_VERTEX) * BYTES_PER_FLOAT
    var coords = floatArrayOf(
            // positions, normals, texture, colors
            -0.2f, -0.2f, 0.2f, 1.0f,
                0.0f, 0.0f, 1.0f, 0.0f,
                   0.0f, 1.0f,
                      1.0f, 0.0f, 0.0f, 1.0f,
            0.2f, -0.2f, 0.2f, 1.0f,
                0.0f, 0.0f, 1.0f, 0.0f,
```

```
                    1.0f, 1.0f,
                        1.0f, 0.0f, 0.0f, 1.0f,
            0.2f, 0.2f, 0.2f, 1.0f,
               0.0f, 0.0f, 1.0f, 0.0f,
                   1.0f, 0.0f,
                       1.0f, 0.0f, 0.0f, 1.0f,
            -0.2f, 0.2f, 0.2f, 1.0f,
                0.0f, 0.0f, 1.0f, 0.0f,
                    0.0f, 0.0f,
                        1.0f, 0.0f, 0.0f, 1.0f
    )
    val drawOrder = shortArrayOf( // vertices order
            0, 1, 2,     0, 2, 3
    )
}
```

Inside the `init` block, the buffers get defined and initialized, and we also load a texture image.

```
init {
    // initialize vertex byte buffer for shape
    // coordinates
    vertexBuffer = ByteBuffer.allocateDirect(
            coords.size * BYTES_PER_FLOAT).apply{
        order(ByteOrder.nativeOrder())
    }.asFloatBuffer().apply {
        put(coords)
        position(0)
    }

    // initialize byte buffer for the draw list
    drawListBuffer = ByteBuffer.allocateDirect(
            drawOrder.size * BYTES_PER_SHORT).apply {
        order(ByteOrder.nativeOrder())
    }.asShortBuffer().apply {
        put(drawOrder)
        position(0)
    }

    if (vertBuf > 0 && idxBuf > 0) {
        GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER,
            vertBuf)
        GLES20.glBufferData(GLES20.GL_ARRAY_BUFFER,
            vertexBuffer.capacity() * BYTES_PER_FLOAT,
            vertexBuffer, GLES20.GL_STATIC_DRAW)

        GLES20.glBindBuffer(
            GLES20.GL_ELEMENT_ARRAY_BUFFER, idxBuf)
        GLES20.glBufferData(
            GLES20.GL_ELEMENT_ARRAY_BUFFER,
            drawListBuffer.capacity() *
                BYTES_PER_SHORT,
            drawListBuffer, GLES20.GL_STATIC_DRAW)
```

```
        GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, 0)
        GLES20.glBindBuffer(
            GLES20.GL_ELEMENT_ARRAY_BUFFER, 0)
    } else {
        // TODO: handle error
    }

    // Load the texture
    textureDataHandle =
        MyGLRenderer.loadTexture(context,
        R.drawable.myImage)
}
```

The `draw()` callback is used to draw the buffers including the texture. Inside the following snippet, we also close the class.

```
fun draw() {
    // Add program to OpenGL ES environment
    GLES20.glUseProgram(program!!)

    // get handle to vertex shader's vPosition member
    val positionHandle =
        GLES20.glGetAttribLocation(program,
        "vPosition")
    // Enable a handle to the vertices
    GLES20.glEnableVertexAttribArray(positionHandle)
    // Prepare the coordinate data
    GLES20.glVertexAttribPointer(positionHandle,
        COORDS_PER_VERTEX,
        GLES20.GL_FLOAT, false,
        VERTEX_STRIDE, vertexBuffer)

    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    // Buffer offsets are a little bit strange in the
    // Java binding - For the other arrays we create
    // a new view and then reset the vertex array
    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    // get handle to vertex shader's vNorm member
    vertexBuffer.position(COORDS_PER_VERTEX)
    val normBuffer = vertexBuffer.slice()
        // create a new view
    vertexBuffer.rewind()
       // ... and rewind the original buffer
    val normHandle =
       GLES20.glGetAttribLocation(program, "vNorm")
    if(normHandle >= 0) {
       // Enable a handle to the vertices
       GLES20.glEnableVertexAttribArray(normHandle)
       // Prepare the coordinate data
```

```
    GLES20.glVertexAttribPointer(normHandle,
        COORDS_PER_VERTEX,
        GLES20.GL_FLOAT, false,
        VERTEX_STRIDE, normBuffer)
}

// get handle to vertex shader's textureCoords
vertexBuffer.position(COORDS_PER_VERTEX +
    NORMS_PER_VERTEX)
val textureBuffer = vertexBuffer.slice()
    // create a new view
vertexBuffer.rewind()
    // ... and rewind the original buffer
val textureHandle =
    GLES20.glGetAttribLocation(program,
    "vTexture")
if(textureHandle >= 0) {
    // Enable a handle to the texture coords
    GLES20.glEnableVertexAttribArray(
        textureHandle)
    // Prepare the coordinate data
    GLES20.glVertexAttribPointer(textureHandle,
        COORDS_PER_VERTEX,
        GLES20.GL_FLOAT, false,
        VERTEX_STRIDE, textureBuffer)
}

// get handle to vertex shader's vColor member
vertexBuffer.position(COORDS_PER_VERTEX +
    NORMS_PER_VERTEX + TEXTURE_PER_VERTEX)
val colorBuffer = vertexBuffer.slice()
    // create a new view
vertexBuffer.rewind()
    // ... and rewind the original buffer
val colorHandle =
    GLES20.glGetAttribLocation(program, "vColor")
if(colorHandle >= 0) {
    // Enable a handle to the vertices
    GLES20.glEnableVertexAttribArray(colorHandle)
    // Prepare the coordinate data
    GLES20.glVertexAttribPointer(colorHandle,
        COLORS_PER_VERTEX,
        GLES20.GL_FLOAT, false,
        VERTEX_STRIDE, colorBuffer)
}
textureUniformHandle =
    GLES20.glGetUniformLocation(program,
    "texture")
if(textureHandle >= 0) {
    // Set the active texture unit to
    // texture unit 0.
    GLES20.glActiveTexture(GLES20.GL_TEXTURE0)
    // Tell the texture uniform sampler to use
```

```
        // this texture in the shader by binding to
        // texture unit 0.
       GLES20.glUniform1i(textureUniformHandle, 0)
    }

    // Draw the plane
    GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER,
        vertBuf)
    GLES20.glBindBuffer(
        GLES20.GL_ELEMENT_ARRAY_BUFFER, idxBuf)
    GLES20.glDrawElements(GLES20.GL_TRIANGLES,
            drawListBuffer.capacity(),
            GLES20.GL_UNSIGNED_SHORT, 0)

    GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, 0)
    GLES20.glBindBuffer(
        GLES20.GL_ELEMENT_ARRAY_BUFFER, 0)

    // Disable vertex array
    GLES20.glDisableVertexAttribArray(
        positionHandle)
    if(normHandle >= 0)
        GLES20.glDisableVertexAttribArray(
        normHandle)
    if(textureHandle >= 0)
        GLES20.glDisableVertexAttribArray(
        textureHandle)
    if(colorHandle >= 0)
        GLES20.glDisableVertexAttribArray(
        colorHandle)
  }
}
```

User Input

To respond to user touch events, all you have to do is overwrite the function `onTouchEvent( e: MotionEvent ) : Boolean { ... }`. The `MotionEvent` you receive is able to tell many things.

- *Touch events*: Touch down and touch up
- *Move events*: Moving while touched
- *Pointer events*: Second, third, ... finger touching

Besides that, a couple more events are registered; see the online documentation of `MotionEvent`.

If we want to listen to touch and move events, the minimum implementation for such a listener reads as follows:

```
override
fun onTouchEvent(event: MotionEvent): Boolean {
    var handled = true
```

```
    when(event.actionMasked) {
        MotionEvent.ACTION_DOWN -> {
            Log.e("LOG","Action: ACTION_DOWN " +
                  event.toString())
        }
        MotionEvent.ACTION_UP -> {
            Log.e("LOG","Action: ACTION_UP " +
                  event.toString())
        }
        MotionEvent.ACTION_MOVE -> {
            Log.e("LOG","Action: MOVE " +
                  event.toString())
        }
        else -> handled = false
    }
    return handled || super.onTouchEvent(event)
}
```

You can see that we return `true` when our listener handles events. If we receive an `ACTION_DOWN` event, returning `true` is important; otherwise, both move and up actions will be ignored.

> **Note** We check the `actionMasked` accessor of the event object, not the `action` accessor as is suggested often. The reason for this is that the `action` accessor might contain additional bits if a multitouch event happens. The masked variant is just more reliable.

UI Design with Movable Items

If your app needs movable UI elements, you should use the `FrameLayout` class or a subclass of it. We want to take care of UI element positions ourselves and don't want a layout class to interfere with that. The `FrameLayout` class does not position its elements dynamically, so using it is a good choice for that kind of positioning.

If you want your view to be movable, you create a subclass and overwrite its `onTouchEvent()` method. For example, say you have an `ImageView` and want to have it movable. For that aim, create a subclass as follows:

```
class MyImageView : ImageView {
  constructor(context: Context)
      : super(context)
  constructor(context: Context, attrs: AttributeSet)
      : super(context, attrs)

  var dx : Float = 0.0f
  var dy : Float = 0.0f

  override
  fun onTouchEvent(event: MotionEvent): Boolean {
      var handled = true
```

```
        when(event.actionMasked) {
            MotionEvent.ACTION_DOWN -> {
                //Log.e("LOG","Action: ACTION_DOWN " +
                //event.toString())
                dx = x - event.rawX
                dy = y - event.rawY
            }
            MotionEvent.ACTION_UP -> {
                //Log.e("LOG","Action: ACTION_UP " +
                //event.toString())
            }
            MotionEvent.ACTION_MOVE -> {
                //Log.e("LOG","Action: MOVE " +
                //event.toString())
                x = event.rawX + dx
                y = event.rawY + dy
            }
            else -> handled = false
        }
        return handled || super.onTouchEvent(event)
    }
}
```

If instead you want to have moving handled by a single point inside your code, you can do one of two things.

- You can create a base class for movable views and let all UI elements inherit from it.
- You can add the touch event listener to the layout container. You then, however, have to provide some logic to find the touched UI element by checking pixel coordinate bounds.

Menus and Action Bars

Menus and action bars are important user interface elements if you want to present your user with a list of selectable choices. In the following sections, we present different kinds of menus and explain when and how to use them.

Options Menu

The options menu shows up inside the app bar. Android Studio will help you to start developing an app with an app bar. If you want to do that yourself, inside your layout add the following:

```
<android.support.design.widget.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:theme="@style/AppTheme.AppBarOverlay">
```

```
    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        app:popupTheme="@style/AppTheme.PopupOverlay"/>
</android.support.design.widget.AppBarLayout>
```

Inside the activity's `onCreate(...)` method, we need to tell Android OS that we are using an app bar. To do so, register the app bar via the following:

```
setSupportActionBar(toolbar)
```

Also, in the activity, overwrite `onCreateOptionsMenu(...)` to create the menu, and overwrite `onOptionsItemSelected(...)` to listen to menu click events.

```
override
fun onOptionsItemSelected(item: MenuItem): Boolean {
  when (item.getItemId()) {
      menu_item1 -> {
          Toast.makeText(this,"Item 1",
                Toast.LENGTH_LONG).show()
          return true
      }
      menu_item2 -> {
          Toast.makeText(this,"Item 2",
                Toast.LENGTH_LONG).show()
          return true
      }
      else -> return
            super.onOptionsItemSelected(item)
  }
}

override
fun onCreateOptionsMenu(menu: Menu): Boolean {
  val inflater = menuInflater
  inflater inflate(R.menu.my_menu, menu)
  return true
}
```

What is left is the definition of the menu itself. Inside `res/menu`, add an XML file called `my_menu.xml` with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android=
      "http://schemas.android.com/apk/res/android">
  <item android:id="@+id/menu_item1"
      android:title="@string/title_item1"/>
  <item android:id="@+id/menu_item2"
      android:title="@string/title_item2"/>
</menu>
```

Here, the `<item>` element accepts more attributes; see the online documentation for more information (search for *Android Menu Resource*). Particularly interesting is the `android:showAsAction` attribute. Setting it to `ifRoom` allows for prominently placing the menu item inside the action bar as a separate action element.

Context Menu

A context menu shows up after a long tap on a registered view. To do this registration, inside the activity call `registerForContextMenu()` and provide the view as an argument.

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

This can be done several times if you want the context menu to show up for several views.

Once a context menu is registered, you define it by overwriting `onCreateContextMenu(...)` inside the activity, and furthermore you overwrite `onContextItemSelected` to listen to menu select events. Here's an example:

```
override
fun onCreateContextMenu(menu: ContextMenu, v: View,
                        menuInfo: ContextMenuInfo?) {
    super.onCreateContextMenu(menu, v, menuInfo)
    val inflater = menuInflater
    inflater.inflate(R.menu.context_menu, menu)
}

override
fun onContextItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        ctxmenu_item1 -> {
            Toast.makeText(this,"CTX Item 1",
                  Toast.LENGTH_LONG).show()
        }
        ctxmenu_item2 -> {
            Toast.makeText(this,"CTX Item 2",
            Toast.LENGTH_LONG).show()
        }
        else -> return
              super.onContextItemSelected(item)
    }
    return true
}
```

The XML definition goes to a standard menu XML file, for example `res/context_menu.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android=
    "http://schemas.android.com/apk/res/android">
  <item android:id="@+id/ctxmenu_item1"
        android:title="@string/ctxtitle_item1"/>
  <item android:id="@+id/ctxmenu_item2"
        android:title="@string/ctxtitle_item2"/>
</menu>
```

It is also possible to open a context menu programmatically by using `openContextMenu( someView )` inside an activity.

Contextual Action Mode

The *contextual action mode* signs responsible for a context-relative app bar. While the app bar is static to the app, a contextual action mode is view specific. What you have to do for this kind of context menu is the following:

1. Implement the `ActionMode.Callback` interface.
   - Inside `onCreateActionMode(...)` create the menu, similar to `onCreateContextMenu()` shown previously for the standard context menu.
   - Inside `onPrepareActionMode(...)` return `false`, unless you need special preparation steps.
   - Implement `onActionItemClicked(...)` to listen to touch events.
2. Create a menu XML resource file, as for the standard context menu.
3. Inside your code, open the contextual action mode by calling `startActionMode( theActionModeCallback )`.

Pop-up Menus

While context-related menus are mostly for settings that are not subject to be changed often, menus that belong to a front-end workflow are best implemented as pop-up menus. Pop-ups usually show up as a result of the user interacting with a certain view, so pop-up menus are assigned to UI elements.

Showing the pop-up menu of a view after some user action can be as easy as calling a function.

```
fun showPopup(v: View) {
   PopupMenu(this, v).run {
     setOnMenuItemClickListener { menuItem ->
       Toast.makeText(this@TheActivity,
             menuItem.toString(),
             Toast.LENGTH_LONG).show()
       true
     }
```

```
     menuInflater.inflate(popup, menu)
     show()
   }
}
```

As usual, you also need to define the menu inside the `res/menu` resources. For the example, it's a file called `popup.xml` (as shown in the first argument to `inflate(...)`).

Progress Bars

Showing progress bars is a good way to improve the user experience for tasks that are taking a couple of seconds. To implement progress bars, add a `ProgressBar` view to your layout. It doesn't matter whether you do that inside the XML layout file or from inside the Kotlin program. In XML you write the following:

```
<ProgressBar
    android:id="@+id/progressBar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
/>
```

for an indeterminate bar (if you can't tell the progress as a percentage value) or the following for a determinate progress bar (you know the percentage while updating the progress bar).

```
<ProgressBar
    android:id="@+id/progressBar"
    style="@android:style/Widget.ProgressBar.Horizontal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:progress="0"
    />
```

Inside the code you toggle the visibility of an indeterminate progress bar by using this:

```
progressBar.visibility = View.INVISIBLE
// or .. = View.VISIBLE
```

To set the progress value for a determinate progress bar, see the following example:

```
with(progressBar) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O)
        min = 0
    max = 100
}
var value = 0
Thread {
    while(value < 100) {
        value += 1
        Thread.sleep(200)
        runOnUiThread {
```

```
            progressBar.progress = value
        }
    }
    progressBar.visibility = View.INVISIBLE
}.start()
```

Here, we use a background thread to simulate the longer-running task. In a real-world app, there will be something more sensible happening in the background. And maybe you use an `AsyncTask` instead of a thread.

Working with Fragments

A *fragment* is a reusable, modular part of an activity. If you develop a nontrivial app without fragments, you will have a number of activities calling each other. The problem is that on devices with larger screens, this is might not be the optimal solution if the activities stay in close relation to each other. Consider, for example, a list of some items in one activity and a details view of a selected item in another activity. On a small device it is perfectly acceptable to start the details activity once the user clicks an item in the list activity. On a larger screen, however, it might improve the user experience if both views, the list and the details, show up beside each other.

That is where fragments come handy. Instead of switching between activities, you create several fragments that are part of the *same* activity. Then, depending on the device, you choose either the one-pane view or the two-pane view.

Developing for fragments is easy if you perform a transition from an app that consists only of activities. Fragments have a lifecycle just as activities do, and the lifecycle callbacks are similar if not the same as for activities. This is where the story gets a little bit complicated, though, because the lifecycle of the container activity and the lifecycles of the contained fragments are connected to each other, and fragments also exhibit a dedicated callstack behavior. The online documentation for fragments gives you a detailed reference for all fragment-related issues; here in this section we limit the survey to the basic aspects of creating and using fragments.

Creating Fragments

To create fragments, you have two options: either you specify `Fragment` elements inside a layout file or you add fragments programmatically from inside the Kotlin code.

To use the XML way of adding fragments to your app, inside a layout file you identify appropriate places where to add fragments and write the following:

```
<fragment android:name=
      "com.example.android.fragments.TheFragment"
    android:id="@+id/fragment_id1"
    android:layout_weight="..."
    android:layout_width="..."
    android:layout_height="..." />
```

Layout parameters are to be chosen according to your app's layout needs. For different devices, more precisely different screen sizes, you can provide several distinct layout files that contain varying numbers of fragments at varying places.

For the fragment class, designated by the `name` attribute in the XML file, start with a bare minimum like the following:

```
import android.support.v4.app.Fragment
...
class MyFragment : Fragment() {
  override
  fun onCreateView(inflater: LayoutInflater?,
        container: ViewGroup?,
        savedInstanceState: Bundle?): View? {
      return inflater!!.inflate(
          my_fragment, container, false)
  }
}
```

Add a layout file called `my_fragment.xml` to the `res/layout` resource folder.

To add fragments from inside your Kotlin code, you identify a layout container or `ViewGroup` where to place a fragment and then use a fragment manager to perform that insertion.

```
with(supportFragmentManager.beginTransaction()) {
    val fragment = MyFragment()
    add(fragm_container.id, fragment, "fragmTag")
    val fragmentId = fragment.id // can use that later...
    commit()
}
```

This can happen, for example, inside the activity's `onCreate()` callback or more dynamically at any other suitable place. `fragm_container` is, for example, a `<FrameLayout>` element inside the layout XML.

> **Note** A fragment gets its ID from the fragment manager while adding it inside a transaction. You cannot use it before that, and you cannot provide your own ID if you add fragments in Kotlin code.

Handling Fragments from Activities

Handling fragments from inside activities includes the following:

- Adding fragments, as shown earlier in the "Creating Fragments" section
- Getting references to fragments, given an ID or a tag
- Handling the back stack
- Registering listeners for lifecycle events

For all those needs, you use `getSupportFragmentManager()`, which gives you the fragment manager that is capable of doing all that, or in Kotlin simply just use the following accessor to fetch a reference:

```
supportFragmentManager
```

> **Note** There is also a `fragmentManager` without "support" in the name. That one points to the framework's fragment manager as opposed to the support library fragment manager. Using the support fragment manager, however, improves the compatibility with older API levels.

Communicating with Fragments

Activities can communicate with their fragments by using the fragment manager and finding fragments based on their ID or tag.

```
val fragm = supportFragmentManager.
      findFragmentByTag("fragmTag")
// or val fragm = supportFragmentManager.
//          findFragmentById(fragmId)
```

But fragments also can talk to their activity. This might indicate a poor application design because fragments should be self-contained entities. If you still need it, you can use `getActivity()` inside the fragment class, or in Kotlin simply use this to access it:

```
activity
```

From there a fragment can even fetch references to other fragments.

App Widgets

App widgets are a dedicated type of application that show informational messages and/or controllers in other apps, especially the home screen. App widgets get implemented as special broadcast receivers and as such are subject to being killed by the Android OS once the callback methods have done their work and a couple of seconds have passed. If you need to have a longer process run, consider starting services from inside the app widgets.

To start creating an app widget, write the following in `AndroidManifest.xml` as a child element of `<application>`:

```
<receiver android:name=".ExampleAppWidgetProvider" >
    <intent-filter>
        <action android:name="android.appwidget.action.
 APPWIDGET_UPDATE" />
    </intent-filter>
    <meta-data android:name="android.appwidget.provider"
               android:resource="@xml/
 example_appwidget_info" />
</receiver>
```

Next create metadata inside the resources. Create a new file called `example_appwidget_info.xml` inside `res/xml` and write the following in it:

```
<appwidget-provider xmlns:android=
      "http://schemas.android.com/apk/res/android"
    android:minWidth="40dp"
    android:minHeight="40dp"
    android:updatePeriodMillis="86400000"
    android:resizeMode="horizontal|vertical"
    android:widgetCategory="home_screen">
</appwidget-provider>
```

What is left is creating the broadcast listener class. For this aim, it is easiest to inherit from class `android.appwidget.AppWidgetProvider`. For example, write the following:

```
class ExampleAppWidgetProvider : AppWidgetProvider() {
  override
  fun onUpdate(context: Context,
        appWidgetManager: AppWidgetManager,
        appWidgetIds:IntArray) {
      // Perform this loop procedure for each App
      // Widget that belongs to this provider
      for(appWidgetId in appWidgetIds) {
          // This is just an example, you can do other
          // stuff here...
          // Create an Intent to launch MainActivity
          val intent = Intent(context,
                MainActivity::class.java)
          val pendingIntent = PendingIntent.
                getActivity(context, 0, intent, 0)

          // Attach listener to the button
          val views =
                RemoteViews(context.getPackageName(),
                R.layout.appwidget_provider_layout)
          views.setOnClickPendingIntent(
                R.id.button, pendingIntent)

          // Tell the AppWidgetManager to perform an
          // update on the app widget
          appWidgetManager.updateAppWidget(
                appWidgetId, views)
      }
  }
}
```

Because an *app widget provider* can serve several app widgets, we have to go through a loop inside the setup. Since this needs a layout file `ciappwidget_provider_layout.xml`, we create this file inside `res/layout` and write the following:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
   xmlns:android=
         "http://schemas.android.com/apk/res/android"
```

```
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="horizontal">

   <Button
      android:id="@+id/button"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="Go"/>
</RelativeLayout>
```

> **Note** Not all layout containers and views are allowed inside the layout file for an app widget. You can use one of these: `FrameLayout`, `LinearLayout`, `RelativeLayout`, `GridLayout`, `AnalogClock`, `Button`, `Chronometer`, `ImageButton`, `ImageView`, `ProgressBar`, `TextView`, `ViewFlipper`, `ListView`, `GridView`, `StackView`, and `AdapterViewFlipper`.

> **Note** The user must still decide to activate an app widget by long-tapping the app icon and then placing it on the home screen. Because not all users know this, the functionality of an app should not depend on whether it gets set as an app widget on the home screen.

App widgets can have a configuration activity attached to them. This special activity gets called once the user tries to place the app widget on the home screen. The user then can be asked for some settings concerning appearance or functioning. To install such an activity, you write the following in `AndroidManifest.xml`:

```
<activity android:name=".ExampleAppWidgetConfigure">
  <intent-filter>
    <action android:name=
        "android.appwidget.action.APPWIDGET_CONFIGURE"/>
  </intent-filter>
</activity>
```

The intent filter shown here is important; it tells the system about this special nature of the activity.

The app widget configuration activity must then be added to the XML configuration file `example_appwidget_info.xml`. Add the following as an additional attribute, using the fully qualified name of the configurator class:

```
android:configure=
      "full.class.name.ExampleAppWidgetConfigure"
```

The configuration activity itself is asked to return the app widget ID as follows:

```
class ExampleAppWidgetConfigure : AppCompatActivity() {
  var awi:Int = 0

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

      awi = intent.extras.getInt(
            AppWidgetManager.EXTRA_APPWIDGET_ID)
      Toast.makeText(this,"" + awi, Toast.LENGTH_LONG).
            show()
      // do more configuration stuff...
  }

  fun goBack(view: View) {
      // just an example...
      val data = Intent()
      data.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
            awi)
      setResult(RESULT_OK, data)
      finish()
  }
}
```

> **Note** The app widget's `onUpdate()` method is not getting called *the first time* the configuration activity exits. It lies in the responsibility of the configuration activity to accordingly call `updateAppWidget()` on the app widget manager if needed.

Drag and Drop

Android supports drag and drop for any kind of UI element and layout.

- The app defines a gesture that defines the start of a drag operation. You are totally free to define this gesture; usual candidates are touch or touch and move operations, but you can also start drag operations programmatically.
- If the drag and drop designates some kind of data transfer, you may assign a data snippet of type `android.content.ClipData` to the drag operation.
- Views that take part of this drag-and-drop operation (this includes *any* drag source and *all* possible drop targets) get a customized `View.OnDragListener` object assigned to.

- While the dragging is in progress, the dragged target gets visually represented by a moving *shadow* object. You are free to define it. It could be a star, a box, or any kind of graphics resembling the drag source. You only have to tell the system how to paint it. The positioning during the drag operation gets automatically handled by the Android OS. Because the drag source stays in place during the drag operation, the layout remains static all the time, and the dragging does not thwart layout operations by the layout manager.
- When the drag shadow enters the area of a possible drop target, the listener gets informed about that, and you can react by, for example, changing the visual appearance of the drop candidate.
- Because all drag sources and possible drop targets get informed about the various dragging states, you are free to visually express that by using different view appearances.
- Once a drop happens, the listener gets informed and you can freely react on such drop events.

It is also possible to not use a dedicated drag-and-drop listener but instead overwrite some specific methods of the views taking part in drag-and-drop operations. The downside of that is you have to use the customized views inside the layout description, which makes it somewhat less readable. Also, from an architectural point of view, the views then know too much of what is happening *to* them, which is an external concern and as such is better handled from outside objects. We therefore follow the listener approach and in the following sections describe what exactly to do for this methodology.

Defining Drag Data

If your drag-and-drop operation defines some kind of data transfer from one object represented by a view to another, you define a `ClipData` object, for example, as follows:

```
val item = ClipData.Item(myView.tag.toString())
val dragData = ClipData(myView.tag.toString(),
      arrayOf(MIMETYPE_TEXT_PLAIN), item)
```

The first argument to the constructor is a user-readable label for the clip datum, the second describes the type of the contents by assigning appropriate MIME types, and the `item` argument represents the datum to be transported, which here is the string given by the `tag` attribute of the view. Other item types are intents and URIs, to be specified in the constructor of `ClipData.Item`.

Defining a Drag Shadow

The drag shadow is the visible element painted underneath your finger while the dragging is in progress. You define the shadow in an object as follows:

```
class DragShadow(val resources: Resources, val resId:Int,
      view: ImageView) : View.DragShadowBuilder(view) {
  val rect = Rect()
```

```
  // Defines a callback that sends the drag shadow
  // dimensions and touch point back to the
  // system.
  override
  fun onProvideShadowMetrics(size: Point, touch: Point) {
      val width = view.width
      val height = view.height

      rect.set(0, 0, width, height)

      // Back to the system through the size parameter.
      size.set(width, height)

      // The touch point's position in the middle
      touch.set(width / 2, height / 2)
  }

  // Defines a callback that draws the drag shadow in a
  // Canvas
  override
  fun onDrawShadow(canvas: Canvas) {
      canvas.drawBitmap(
      BitmapFactory.decodeResource(
            resources, resId),
      null, rect, null)
  }
}
```

This example draws a bitmap resource, but you can do anything you like here.

Starting a Drag

To start a drag, you invoke `startDragAndDrop()` for API level 24 and higher, or you invoke `startDrag()` on everything else, on the view object that serves as a drag source. Here's an example:

```
theView.setOnTouchListener { view, event ->
    if(event.action == MotionEvent.ACTION_DOWN) {
        val shadow = DragShadow(resources,
              R.the_dragging_image, theView)

        val item = ClipData.Item(frog.tag.toString())
        val dragData = ClipData(frog.tag.toString(),
              arrayOf(MIMETYPE_TEXT_PLAIN), item)

        if (Build.VERSION.SDK_INT >=
              Build.VERSION_CODES.N) {
          theView.startDragAndDrop(dragData, shadow,
                null, 0)
```

```
        } else {
          theView.startDrag(dragData, shadow,
                null, 0)
        }
    }
    true
}
```

If the drag operation is *not* associated with a data type, you can just as well let `dragData` be `null`. In this case, you don't have to build a `ClipData` object.

Listening to Drag Events

The complete set of events occurring during a drag-and-drop operation is governed by a drag-and-drop listener. Here's an example:

```
class MyDragListener : View.OnDragListener {
  override
  fun onDrag(v: View, event: DragEvent): Boolean {
      var res = true
      when(event.action) {
          DragEvent.ACTION_DRAG_STARTED -> {
              when(v.tag) {
                  "DragSource" -> { res = false
                        /*not a drop receiver*/ }
                  "OneTarget" -> {
                      // could visibly change
                      // possible drop receivers
                  }
              }
          }
          DragEvent.ACTION_DRAG_ENDED -> {
              when(v.tag) {
                  "OneTarget" -> {
                      // could visibly change
                      // possible drop receivers
                  }
              }
          }
          DragEvent.ACTION_DROP -> {
              when(v.tag) {
                  "OneTarget" -> {
                      // visually revert drop
                      // receiver ...
                  }
              }
              Toast.makeText(v.context, "dropped!",
                    Toast.LENGTH_LONG).show()
          }
```

```
            DragEvent.ACTION_DRAG_ENTERED -> {
                when(v.tag) {
                    "OneTarget" -> {
                        // could visibly change
                        // possible drop receivers
                    }
                }
            }
            DragEvent.ACTION_DRAG_EXITED -> {
                when(v.tag) {
                    "OneTarget" -> {
                        // visually revert drop
                        // receiver ...
                    }
                }
            }
        }
        return res
    }
}
```

You can see that we are listening to drag start and end events, to the shadow entering or exiting a possible drop area, and to drop events. As pointed out, how you react to all these events is totally up to you.

> **Note** In the example we use a `tag` attribute assigned to a view to identify the view as part of a drag-and-drop operation. In fact, you can also use the ID or any other way you might think of.

What is left is inside the activity's `onCreate()` callback; you register the listener to all views that participate in a drag-and-drop operation.

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

  theView.setOnTouchListener ...

  val dragListener = MyDragListener()
  theView.setOnDragListener(dragListener)
  otherView.setOnDragListener(dragListener)
  ...
}
```

Multitouch

Multitouch events in the Android OS are surprisingly easy to handle. All you need is to overwrite the `onTouchEvent()` method of a `View` or a `ViewGroup` element. Inside the `onTouchEvent()`, you fetch the masked action and act upon it.

```
frog.setOnTouchListener { view, event ->
    true
}
```

> **Note** In older versions of Android, you usually dispatch on the action as in `event.action`. With multitouch gestures, it is better to act on the `maskedAction`.

Inside the listener, you get the masked action by `event.actionMasked` and pass it to a `when() { .. }` statement.

The magic now lies in this listener being invoked for all fingers (here called *pointers*) consecutively. To find out how many fingers currently are registered, you use `event.pointerCount`, and if you want to know which finger the event belongs to you, use `val index = event.actionIndex`. A starting point is thus as follows:

```
theView.setOnTouchListener { view,event ->
  fun actionToString(action:Int) : String = mapOf(
    MotionEvent.ACTION_DOWN to "Down",
    MotionEvent.ACTION_MOVE to "Move",
    MotionEvent.ACTION_POINTER_DOWN to "Pointer Down",
    MotionEvent.ACTION_UP to "Up",
    MotionEvent.ACTION_POINTER_UP to "Pointer Up",
    MotionEvent.ACTION_OUTSIDE to "Outside",
    MotionEvent.ACTION_CANCEL to "Cancel").
          getOrDefault(action,"")

  val action = event.actionMasked
  val index = event.actionIndex
  var xPos = -1
  var yPos = -1
  Log.d("LOG", "The action is " +
        actionToString(action))

  if (event.pointerCount > 1) {
    Log.d("LOG", "Multitouch event")
    // The coordinates of the current screen contact,
    // relative to the responding View or Activity.
    xPos = event.getX(index).toInt()
    yPos = event.getY(index).toInt()
  } else {
    // Single touch event
```

```
    Log.d("LOG", "Single touch event")
    xPos = event.getX(index).toInt()
    yPos = event.getY(index).toInt()
  }

  // do more things...

    true
}
```

Picture-in-Picture Mode

Starting with Android 8.0 (API level 26), there exists a picture-in-picture mode where an activity gets shrunk and pinned to an edge of the screen. This is especially useful if the activity plays a video and you want the video to keep playing while another activity shows up.

To enable the picture-in-picture mode, inside `AndroidManifest.xml` add the following attributes to `<activity>`:

```
android:resizeableActivity="true"
android:supportsPictureInPicture="true"
android:configChanges=
    "screenSize|smallestScreenSize|
     screenLayout|orientation"
```

Then, wherever feasible in your app, you start the picture-in-picture mode by using this:

```
enterPictureInPictureMode()
```

You might want to change and later revert the layout if the picture-in-picture mode gets entered or exited. To do so, overwrite `onPictureInPictureModeChanged( isInPictureInPictureMode : Boolean, newConfig : Configuration )` and react accordingly.

Text to Speech

The text to speech framework allows for text to be converted to audio, either directly sent to the audio hardware or sent to a file. Using the corresponding `TextToSpeech` class is easy, but you should make sure all necessary resources are loaded. For this aim, an intent with action `TextToSpeech.Engine.ACTION_CHECK_TTS_DATA` should be fired. It is expected to return `TextToSpeech.Engine.CHECK_VOICE_DATA_PASS`; if it does not, call another intent with action `TextToSpeech.Engine.ACTION_INSTALL_TTS_DATA` to let the user install text to speech data.

An example activity doing all that reads as follows:

```
class MainActivity : AppCompatActivity() {
  companion object {
      val MY_DATA_CHECK_CODE = 42
  }

  var tts: TextToSpeech? = null
```

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

      val checkIntent = Intent()
      checkIntent.action = TextToSpeech.Engine.
            ACTION_CHECK_TTS_DATA
      startActivityForResult(checkIntent,
            MY_DATA_CHECK_CODE)
  }

  fun go(view: View) {
       tts?.run {
           language = Locale.US
           val myText1 = "Did you sleep well?"
           val myText2 = "It's time to wake up."
           speak(myText1, TextToSpeech.QUEUE_FLUSH, null)
           speak(myText2, TextToSpeech.QUEUE_ADD, null)
       }
  }

  override
  fun onActivityResult(requestCode: Int,
      resultCode: Int, data: Intent) {
  if (requestCode == MY_DATA_CHECK_CODE) {
    if (resultCode ==
          TextToSpeech.Engine.CHECK_VOICE_DATA_PASS) {
      // success, create the TTS instance
      tts = TextToSpeech(this, { status ->
        // do s.th. if you like
      })
    } else {
      // data are missing, install it
      val installIntent = Intent()
      installIntent.action =
          TextToSpeech.Engine.ACTION_INSTALL_TTS_DATA
      startActivity(installIntent)
    }
   }
  }
}
```

This example also contains a `go()` method, which for example could be triggered by a button press. I will produce some speech and send it immediately to the loudspeakers.

If instead you want to write the audio to a file, use the `tts.synthesizeToFile()` method. You can find more details in the online documentation of `TextToSpeech`.

Chapter 10

Development

This chapter covers issues closer to development matters, compared to the previous chapters. The topics we will be talking about here are less tightly coupled to specific Android OS APIs. It is more our concern here to find out how technical requirements can best be accomplished using Kotlin methodologies.

The chapter also has a section that covers transcribing Kotlin code to JavaScript code that can serve `WebView` widgets.

Writing Reusable Libraries in Kotlin

Most tutorials you will find on the Web are about activities, services, broadcast receivers, and content providers. These components are reusable in the sense that you can more or less easily extract them from a project and copy them to another project. The encapsulation in the Android OS has reached an elaborate stage, which makes this reuse possible. On a lower level, however, in some cases the libraries or APIs that are provided inside Android might not suit all your needs, so you might be tempted to develop such libraries yourself and then copy the sources from project to project wherever feasible.

Surely, such a copying on the source code level does not fit well into modern methodologies for reusable libraries; just think about maintenance and versioning issues that introduce a lot of boilerplate efforts. The best thing is to design such reusable libraries as dedicated development artifacts. They then can be easily reused from different projects.

In the following sections, we develop a rudimentary regular expression library, serving as a conceptual basis for your own library projects.

Starting a Library Module

Library projects are projects containing one or more modules. With Android Studio open, create a new project and make sure it has Kotlin support enabled. Then, inside the new project go to New ➤ New module and choose Android Library.

P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_10

> **Note** An Android library is more than just a collection of classes. It may also contain resources and configuration files. For our purposes, we'll just look at the classes. From a development perspective, these additional possibilities don't hurt, and you just can ignore them. For projects using the Android Library type, however, this gives you many more possibilities for future extensions, compared to using just JAR files.

Creating the Library

Inside the library module, create a new Kotlin class and inside write the following:

```
package com.example.regularexpressionlib

infix operator fun String.div(re:String) :
      Array<MatchResult> =
  Regex(re).findAll(this).toList().toTypedArray()

infix operator fun String.rem(re:String) :
      MatchResult? =
  Regex(re).findAll(this).toList().firstOrNull()

operator fun MatchResult.get(i:Int) =
  this.groupValues[i]

fun String.onMatch(re:String, func: (String)-> Unit)
      : Boolean =
this.matches(Regex(re)).also { if(it) func(this) }
```

What these four operators and functions do is not less than allowing us to write `searchString/regExpString` to search for regular expression matches and `searchString % regExpString` to search for the first match. In addition, we can use `searchString.onMatch()` to have some block execute only if there is a match.

This listing is different from all the listings we have seen so far in this book. First, you can see that we don't have any class here. This is possible because Kotlin knows the concept of a file artifact. Behind the scenes it generates a hidden class based on the package name. Any client of the library that imports it via `import com.example.regularexpressionlib.*` can act as if it performed a static import of all these functions in Java.

`infix operator fun String.div( re:String )` defines a division operator for strings. Such a division is not possible in the standard, so there is no clash with Kotlin built-in operators. It uses the `Regex` class from the Kotlin libraries to find all occurrences of a regular expression in a search string and convert it to an array, so we can later use the `[ ]` operator to access the results by index. `infix operator fun String.rem( re:String )` does almost the same, but it defines the `%` operator for strings, performs a regular expression search, and takes only the first result or returns `null` if no result exists.

`operator fun MatchResult.get(i:Int) = ...` is an extension of the `MatchResult` returned by the previous operators. It allows for accessing the groups of a match by index. Say if, for example, you searched for *(el)" in- side "Hello Nelo"*, you can write `("Hello Nelo" / "(e.)") [0][1]` to get the first group of the first match, in this case the *el* from *Hello*

Testing the Library

We need a way to test the library while developing it. Unfortunately, Android Studio 3.0 does not allow for something like a `main()` function. The only thing we can do is to create a unit test, and for our case such a unit test could read as follows:

```
import org.junit.Assert.*
import org.junit.Test
...

class RegularExpressionTest {
  @Test
  fun proof_of_concept() {
      assertEquals(1, ("Hello Nelo" / ".*el.*").size)
      assertEquals(2, ("Hello Nelo" / ".*?el.*?").size)

      var s1:String = ""
      ("Hello Nelo" / "e[lX]").forEach {
          s1 += it.groupValues
      }
      assertEquals("[el][el]", s1)

      var s2: String = ""
      ("Hello Nelo" / ".*el.*").firstOrNull()?.run {
          s2 += this[0]
      }
      assertEquals("Hello Nelo", s2)
      assertEquals("el",
            ("Hello Nelo" % ".*(el).*")?.let{ it[1] } )

      assertEquals("el",
            ("Hello Nelo" / ".*(el).*")[0][1])

      var match1: Boolean = false
      "Hello".onMatch(".*el.*") {
         match1 = true
      }
      assertTrue(match1)
  }
}
```

You can then run this test like any other unit test using Android Studio's context menu. Note that at early stages of the development you can add `println()` statements to the test to print some information on the test console while the test runs.

Using the Library

Once you invoke Build ➤ Rebuild Project, you can find the Android library inside this folder of the module.

```
build/outputs/aar
```

To use it from clients, create a new module in the client project via New ➤ New Module, choose Import .JAR/.AAR Package, and navigate to the `.aar` file generated by the library project.

> **Caution** This procedure copies the `.aar` file. If you have a new version of the library, you can either remove the library project inside the client project and import it again or copy the `.aar` file manually from the library project to the client project.

To use the library in the client, you just add `import com.example.regularexpressionlib.*` to the header, and henceforth you can apply the new matching constructs as shown in the previous test.

Publishing the Library

So far we have used the library locally, which means you have the library project somewhere on your development machine and can use it from other projects on the same machine. You can also publish libraries, meaning make them available for other developers inside a corporate environment if you have a corporate repository at hand or make them truly public for libraries you want to provide to the community.

Unfortunately, the process of publishing libraries is rather complex and involves altering the build files in several places and using third party plug-ins and repository web sites. This makes the process of publishing libraries a complex and brittle task, and a detailed description of one possible publishing process might easily be outdated when you read this book. I therefore ask you to do your own research. Entering *publishing android libraries* in your favorite search engine will readily point you to online resources that will help you. If you find several processes that might suit your needs, a general rule of thumb is to use one that has a large supporting community and is as easy as possible.

Also, make sure for corporate projects you have the allowance to use public repositories if you want to use one of them. If you cannot use public repositories, installing a corporate repository is not an overly complex task. To establish a corporate Maven repository, you can use the software suite Artifactory.

Advanced Listeners Using Kotlin

Whatever kind of app you are creating for Android, at one or the other place or more probably quite often you will have to provide listeners for API function calls. While in Java you have to create classes or anonymous inner classes that implement the listener interface, in Kotlin you can do that more elegantly.

If you have a single abstract method (SAM) class or interface, it is easy. For example, if you want to add an on-click listener to a button, which means you have to provide an implementation of interface `View.OnClickListener`, doing it the Java way looks like this:

```
btn.setOnClickListener(object : View.OnClickListener {
    override fun onClick(v: View?) {
        // do s.th.
    }
})
```

However, since this interface just has one method, you can write it more succinctly, like this:

```
btn.setOnClickListener {
  // do s.th.
}
```

You can let the compiler find out how the interface method should be implemented.

If a listener is not a SAM, that means if it has more than one method, this short notation is not possible any longer. If, for example, you have an `EditText` view and want to add a text change listener, you basically have to write the following even if you are interested only in the `onTextChanged()` callback method.

```
val et = ... // the EditText view
et.addTextChangedListener( object : TextWatcher {
    override fun afterTextChanged(s: Editable?) {
        // ...
    }
    override fun beforeTextChanged(s: CharSequence?,
          start: Int, count: Int, after: Int) {
        // ...
    }
    override fun onTextChanged(s: CharSequence?,
          start: Int, before: Int, count: Int) {
        // ...
    }
})
```

What you could do, however, is extend the `EditText` class in a utility file and add the possibility to provide a simplified text changed listener. To do so, start with such a file, for example, `utility.kt` inside package `com.example` or of course any package of your app. Add the following:

```
fun EditText.addTextChangedListener(l:
      (CharSequence?, Int, Int, Int) -> Unit) {
  this.addTextChangedListener(object : TextWatcher {
      override fun afterTextChanged(s: Editable?) {
      }
      override fun beforeTextChanged(s: CharSequence?,
            start: Int, count: Int, after: Int) {
      }
      override fun onTextChanged(s: CharSequence?,
            start: Int, before: Int, count: Int) {
          l(s, start, before, count)
      }
  })
}
```

This adds the desired method to that class dynamically.

You can now use `import com.example.utility.*` anywhere needed and then write the following, which looks considerably more concise compared to the original construct:

```
val et = ... // the EditText view
et.addTextChangedListener({ s: CharSequence?,
      start: Int, before: Int, count: Int ->
  // do s.th.
})
```

Multithreading

We already talked about multithreading to some extent in Chapter 9. In this section, we just point out what Kotlin on a language level can do to simplify multithreading.

Kotlin contains a couple of utility functions inside its standard library. They help to start threads and timers more easily compared to using the Java API; see Table 10-1.

Table 10-1. Kotlin Concurrency

Name	Parameters	Return	Description
fixedRate-Timer	name: String? daemon: Boolean initialDelay: Long period: Long action: TimerTask.() -> Unit	Timer	Creates and starts a timer object for fixed-rate scheduling. The period and initialDelay parameters are in milliseconds.
fixedRate-Timer	name: String? daemon: Boolean startAt: Date period: Long action: TimerTask.() -> Unit	Timer	Creates and starts a timer object for fixed-rate scheduling. The period parameter is in milliseconds.
timer	name: String? daemon: Boolean initialDelay: Long period: Long action: TimerTask.() -> Unit	Timer	Creates and starts a timer object for fixed-rate scheduling. The period parameter is the time in milliseconds between the end of the previous and the start of the next task.
timer	name: String? daemon: Boolean startAt: Date period: Long action: TimerTask.() -> Unit	Timer	Creates and starts a timer object for fixed-rate scheduling. The period parameter is the time in milliseconds between the end of the previous and the start of the next task.
thread	start: Boolean isDaemon: Boolean contextClassLoader: ClassLoader? name: String? priority: Int block: () -> Unit	Thread	Creates and possibly starts a thread, executing its block. Threads with higher priority are executed in preference to threads with lower priority.

For the timer functions, the `action` parameter is a closure with `this` being the corresponding `TimerTask` object. Using it, you may, for example, cancel the timer from inside its execution block. Threads or timers that have `daemon` or `isDaemon` set to `true` will not prevent the JVM from shutting down when all nondaemonized threads have exited.

By virtue of its general functional abilities, Kotlin does a good job in helping us with concurrency; many of the classes inside `java.util.concurrent` that deal with parallel execution take a `Runnable` or `Callable` as an argument, and in Kotlin you can always replace such SAM constructs via direct `{ ... }` lambda constructs. Here's an example:

```
val es = Executors.newFixedThreadPool(10)
// ...
val future = es.submit({
      Thread.sleep(2000L)
      println("executor over")
      10
  } as ()->Int)
val res:Int = future.get()
```

So, you don't have to write the following as in Java:

```
ExecutorService es = Executors.newFixedThreadPool(10);
// ...
Callable<Integer> c = new Callable<>() {
    public Integer call() {
      try {
        Thread.sleep(2000L);
      } catch(InterruptedException e { }
      System.out.println("executor over");
      return 10;
    };
Future<Integer> f = es.submit(c);
int res = f.get();
```

Note, the cast to `()->Int` is necessary in the Kotlin code, even with Android Studio complaining that it is superfluous. The reason for that is if we didn't do it, the other method with a `Runnable` as an argument gets called instead, and the executor is unable to return a value.

Compatibility Libraries

There is an important and at the beginning not so easy to understand distinction between the Framework API and the compatibility libraries. If you start developing Android apps, you quite often see classes of the same name showing up in different packages. Or even worse, you see classes of different names from different packages seemingly doing the same thing.

Let's take a look at a prominent example. To create activities, either you can subclass `android.app.Activity` or you can subclass `android.support.v7.app.AppCompatActivity`. Looking at examples and tutorials you find on the Web, there seems to be no noticeable difference in usage. In fact, `AppCompatActivity` inherits from `Activity`, so wherever `Activity` is required, you can substitute `AppCompatActivity` for it, and it will compile. So, is there a difference in function? It depends. If you look at the documentation or at the code, you can see that the `AppCompatActivity` allows for adding `android.support.v7.app.ActionBar`, which `android.app.Activity` does not. Instead, `android.app.Activity` allows for adding `android.app.ActionBar`. And this time `android.support.v7.app.ActionBar` does not inherit from `android.app.ActionBar`, so you cannot add `android.support.v7.app.ActionBar` to `android.app.Activity`.

This basically says that if you favor `android.support.v7.app.ActionBar` over `android.app.ActionBar`, you must use `AppCompatActivity` for an activity. Why would one use `android.support.v7.app.ActionBar` instead of `android.app.ActionBar`? The answer is easy: the latter is quite old; it has been available since API level 11. Newer versions of `android.app.ActionBar` cannot break the API to maintain compatibility with older devices. But `android.support.v7.app.ActionBar` can have new functions added; it is much newer and has existed since API level 24.

The magic now works as follows: if you use a device that speaks API level 24 or higher, you can use `android.support.v7.app.AppCompatActivity` and add `android.support.v7.app.ActionBar`. You could also use `android.app.Activity`, but then you cannot add `android.support.v7.app.ActionBar` and instead have to use `android.app.ActionBar`. So, for new devices, it makes sense to use `android.support.v7.app.AppCompatActivity` for your activities if the support library action bar better suits your needs compared to the framework action bar.

How about older devices? You still can use `android.support.v7.app.AppCompatActivity` because it is provided as a library added to the app. So, you also can use the modern `android.support.v7.app.ActionBar` as an action bar and have more functionalities compared to the old `android.app.ActionBar` genuinely provided by the device. And this is actually how the trick goes; by using support libraries, even older devices can take advantage of new functionalities added later! The implementation of the support class internally checks for the device version and provides sensible fallback functionalities to resemble modern devices as much as possible.

The caveat is that you as a developer have to live in two worlds at the same time. You have to explicitly or implicitly use framework classes if there is no other choice, and you have to think about using support library classes if available and if you want to ensure the maximum compatibility with older devices. It is therefore vital to check, before using a class, whether there is also a matching support library class. You might not be happy with this two-world methodology used in Android, and it also means more thinking work to build an app, but that is how Android handles backward compatibility.

You will readily find detailed information about the support libraries if you enter *android support library* in your favorite search engine.

Support libraries get bundled with your app, so they must be declared as dependencies in the build file. If you start a new project in Android Studio, it by default writes inside the module's `build.gradle` file.

```
dependencies {
 ...
  implementation 'com.android.support:appcompat-v7
 :26.1.0'
  implementation 'com.android.support.constraint:
   constraint-layout:1.0.2'
...
}
```

You can see the support library version 7 is available by default, so you can use it right from the start.

Kotlin Best Practices

Development is not only about solving IT-related problems or implementing requirements; you also want to write "good" software. What "good" exactly means in this context is a little blurry, though. A lot of aspects play a role here: quick development, high execution performance, short programs, readable programs, high program stability, and so on. All of them have their merits, and exaggerating any of them will thwart the other aspects.

In fact, you should have all of them in mind, but my experience says to put some emphasis on the following aspects:

- Make programs comprehensive (or expressive). A super-elegant solution that nobody else understands might make you happy, but bear in mind that later maybe other people need to understand your software.
- Keep programs simple. Overly complex solutions are subject to instabilities. Of course, you will not wake up one morning and say, "OK, today I will write a simple program to solve requirement XYZ." Writing simple programs that reliably solve problems is a matter of experience, and it comes with years of practice. But you can always try to constantly get better in writing simple programs. A good starting point is always asking yourself, "Shouldn't there be a simpler solution to this?" for any part of your software, and by looking at the API documentations and the programming language reference in quite some cases, you will find easier solutions doing the same as what you currently have.
- Don't repeat yourself. This principle, commonly referred to as DRY programming, cannot be overemphasized. Whenever you find yourself using Ctrl+C and Ctrl+V to copy some program passages, think of instead using one function or one lambda expression to provide just one place where things are done.
- Do expected things. You can overwrite class methods and operators in Kotlin, and you can dynamically add functions to existing classes, even to such basic classes like `String`. In any case, make sure such extensions work as expected by looking at their names because if they don't, the program is hard to understand.

Kotlin helps with all of these aspects and quite often does a better job than the venerable Java. In the following sections, we point out a couple of Kotlin concepts you can use to make your program short, simple, and expressive. Note that the sum of these concepts is far from being a complete documentation of Kotlin. So for more details, please see the online documentation.

Functional Programming

While functional programming as a development paradigm entered Java with version 8, Kotlin has supported a functional programming style from the beginning. In functional programming, you prefer nonmutable values over variables, avoid state machines, and allow functions as parameters to functions. Also, the lambda calculus allows for passing functions without names. Kotlin provides us with all that.

In Java you have the `final` modifier to express that a variable isn't going to be changed after the first initialization. While most Java developers use `final` modifiers for constants; I barely ever see developers using them in the coding.

```
public class Constants {
  public final static int CALCULATION_PRECISION = 10;
  public final static int MAX_ITERATIONS = 1000;
  ...
}
```

This is a pity since it improves both readability and stability. The temptation to omit it to save a few keystrokes is just too big. In Kotlin the story is different; you say `val` to express that a data object remains constant during its lifetime, and you use `var` if you need a real variable, as shown here:

```
fun getMaxFactorial():Int = 13
fun fact(n:Int):Int {
    val maxFactorial = getMaxFactorial()
    if(n > maxFactorial)
       throw RuntimeException("Too big")
    var x = 1
    for( i in 1.. (n) ) {
         x *= i
    }
    return x
}
val x = fact(12)
System.out.println("12! = ${x}")
```

This short snippet uses `maxFactorial` as a `val`, which means "This is not subject to change." The `x` however is a `var`, and it gets changed after initialization.

We can even avoid `var x` in the snippet for the factorial calculation and replace it with a functional construct. This is another functional imperative: prefer expressions over a statement or a chain of statements. To do so, we use a recursion and write the following:

```
fun fact(n:Int):Int = if(n>getMaxFactorial())
  throw RuntimeException("Too big") else
  if(n > 1) n * fact(n-1) else 1
val x = fact(10)
System.out.println("10! = ${x}")
```

This little factorial calculator is just a short example. With collections, the story gets more interesting. The Kotlin standard library includes a lot of functional constructs you can use to write elegant code. Just to give you a glimpse of all the possibilities, we rewrite the factorial calculator once again and use a `fold` function from the collections package.

```
fun fact(n:Int) = (1..n).fold(1, { acc,i -> acc * i })
System.out.println("10! = ${fact(10)}")
```

For simplicity I removed the range check; if you like, you can add that `if...` check from earlier to the lambda expression inside `{...}`. You see that we even don't have a `val` left; internally the `i` and `acc` get handled as `vals`, though. This can even be shortened one step further. Since all we use is the "times" functionality of type `Int`, we can directly refer to it and write the following:

```
fun fact(n:Int) = (1..n).fold(1, Int::times)
System.out.println("10! = ${fact(10)}")
```

With the other functional constructs from the collections package, you can perform more interesting transformations with sets, lists, and maps. But functional programming is also about passing around functions as objects in your code. In Kotlin you can assign functions to `vals` or `vars` as follows:

```
val factEngine: (acc:Int,i:Int) -> Int =
      { acc,i -> acc * i }
fun fact(n:Int) = (1..n).fold(1, factEngine)
System.out.println("10! = ${fact(10)}")
```

or as follows, which is even shorter since Kotlin under certain circumstances can infer the type:

```
val factEngine = { acc:Int, i:Int -> acc * i }
fun fact(n:Int) = (1..n).fold(1, factEngine)
System.out.println("10! = ${fact(10)}")
```

In this book we are using functional constructs wherever feasible to improve comprehensiveness and conciseness.

Top-Level Functions and Data

While in the Java world it is considered bad style to have too many globally available functions and data, for example by definitions with static scope inside some utility class, in Kotlin this has experienced a renaissance and also looks somewhat more natural. That is because you can declare functions and variables/values in a file outside any class. Still, to use them, you have to import such elements like in `import com.example.global.*` where a file with an arbitrary name inside package `com/example.global` contains no classes but only `fun`, `var`, and `val` elements.

For example, write a file called `common.kt` in `com/example/app/util` and add the following in it:

```
package com.example.app.util

val PI_SQUARED = Math.PI * Math.PI

fun logObj(o:Any?) =
      o?.let { "(" + o::class.toString() + ") " +
                o.toString() } ?: "<null>"
```

Then add more utility functions and constants. To use them, write the following:

```
import com.example.app.util.*
...
val ps = PI_SQUARED
logObj(ps)
```

You should, however, be cautious using that feature; overly using it easily leads to a structural mess. Avoid putting mutable variables in such a scope altogether! You can and should put utility functions and global constants in such a global file.

Class Extensions

Unlike in the Java language, Kotlin allows you to dynamically add methods to classes. To do so, write the following:

```
fun TheClass.newFun(...){ ... }
```

The same works for operators, which allows you to create extensions like `"Some Text" % "magic"` (it is left to your imagination what this does) to such common classes like `String`. You'd implement this particular extension like this:

```
infix operator fun String.rem(s:String){ ... }
```

Just make sure you don't unintentionally overwrite existing class methods and operators. This makes your program unreadable because it does unexpected things. Note that most standard operators like `Double.times()` cannot be overwritten anyway since they are marked final internally.

Table 10-2 describes the operators you can define via `operator fun TheClass.<OPER- ATOR>`.

Table 10-2. Kotlin Operators

Symbol	Translates to	Infix	Default Function
`+a`	`a.unaryPlus()`		Usually does nothing.
`-a`	`a.unaryMinus()`		Negates a number.
`!a`	`a.not()`		Negates a Boolean expression.
`a++`	`a.inc()`		Increments a number.
`a- -`	`a.dec()`		Decrements a number.
`a + b`	`a.plus(b)`	x	Addition.
`a - b`	`a.minus(b)`	x	Subtraction
`a * b`	`a.times(b)`	x	Multiplication.
`a / b`	`a.div(b)`	x	Division.
`a % b`	`a.rem(b)`	x	Remainder after division.
`a . . b`	`a.rangeTo(b)`	x	Defines a range.
`a in b`	`b.contains(a)`	x	Containment check.
`a !in b`	`!b.contains(a)`	x	Not-containment check.
`a[i]`	`a.get(i)`		Indexed access.
`a[i,j,...]`	`a.get(i,j,...)`		Indexed access, normally not used.
`a[i] = b`	`a.set(i,b)`		Indexed setting access.
`a[i,j,...] = b`	`a.set(i,j,...,b)`		Indexed setting access, normally not used.
`a()`	`a.invoke()`		Invocation.
`a(b)`	`a.invoke(b)`		Invocation.
`a(b,c,...)`	`a.invoke(b,c,...)`		Invocation.
`a += b`	`a.plusAssign(b)`	x	Adds to `a`. Must not return a value; instead, you must modify `this`.
`a -= b`	`a.minusAssign(b)`	x	Subtracts from `a`. Must not return a value; instead, you must modify `this`.
`a *= b`	`a.timesAssign()`	x	Multiplies to `a`. Must not return a value; instead, you must modify `this`.
`a /= b`	`a.divAssign(b)`	x	Divides `a` by `b` and then assigns. Must not return a value; instead, you must modify `this`.
`a %= b`	`a.remAssign(b)`	x	Takes the remainder of the division by `b` and then assigns. Must not return a value; instead, you must modify `this`.
`a == b`	`a?.equals(b) ?: (b === null)`	x	Checks equality.

(*continued*)

Table 10-2. (*continued*)

Symbol	Translates to	Infix	Default Function
`a != b`	`!(a?.equals(b) ?: (b === null))`	x	Checks inequality.
`a > b`	`a.compareTo(b) > 0`	x	Comparison.
`a < b`	`a.compareTo(b) < 0`	x	Comparison.
`a >= b`	`a.compareTo(b) >= 0`	x	Comparison.
`a <= b`	`a.compareTo(b) <= 0`	x	Comparison.

To define an extension, for any operator from the table of type `Infix`, you write the following:

```
infix operator fun TheClass.<OPERATOR>( ... ){ ... }
```

Here, the function arguments are the second and any subsequent operands, and `this` inside the function body refers to the first operand. For operators not of type `Infix`, just omit the `infix`.

Defining operators for your own classes certainly is a good idea. Amending standard Java or Kotlin library classes by operators might improve the readability of your code as well.

Named Arguments

By using named arguments as follows:

```
fun person(fName:String = "", lName:String = "",
      age:Int=0) {
   val p = Person().apply { ... }
   return p
}
```

you can make more expressive calls like this:

```
val p = person(age = 27, lName = "Smith")
```

Using parameter names means you don't have to care about argument order, and in many cases you can avoid overloading constructors for various parameter combinations.

Scoping Functions

The scoping functions allow you to structure your code in a way that's different from using classes and methods. Consider, for example, the following code:

```
val person = Person()
person.lastName = "Smith"
person.firstName = "John"
person.birthDay = "2011-01-23"
val company = Company("ACME")
```

While this is valid code, the repetition of `person.` is annoying. Besides, the first four lines are about constructing a person, while the next line has nothing to do with a person. It would be nice if this could be visually expressed, and the repetition could also be avoided. This is a construct in Kotlin, and it reads as follows:

```
val person = Person().apply {
  lastName = "Smith"
  firstName = "John"
  birthDay = "2011-01-23"
}
company = Company("ACME")
```

This looks more expressive compared to the original code. It clearly says construct a person, do something with it, and then do something else. There are five such constructs, and despite being similar, they differ in meaning and usage: `also`, `apply`, `let`, `run`, and `with`. Table 10-3 describes them.

***Table 10-3.** Scoping Functions*

Syntax	What Is `this`	What is it	Returns	Use
`a.also { ... }`	`this` of outer context	`a`	`a`	Use for some crosscutting concern, for example to add logging.
`a.apply { ... }`	`a`	-	`a`	Use for postconstruction object forming.
`a.let { ... }`	`this` of outer context	`a`	Last expression	Use for transformations.
`a.run { ... }`	`a`	-	Last expression	Do some computation using an object, with only side effects. For c better clarity, don't use what it returns.
`with(a) { ... }`	`a`	-	Last expression	Group operations on an object. For better clarity, don't use what it returns.

Using scoping functions greatly improves the expressiveness of your code. I use them often in this book.

Nullability

Kotlin addresses the problem of nullability on a language level, to avoid annoying `NullPointerException` throws. For any variable or constant, the assignment of `null` values is not allowed by default; you have to explicitly declare nullability by adding a `?` at the end as follows:

```
var name:String? = null
```

The compiler then knows that `name` from the example can be `null` and takes various precautions to avoid `NullPointerExceptions`. You, for example, cannot write `name.toUpperCase()`, but you have to use `name?.toUpperCase()` instead, which does the capitalization only if `name` is not `null` and otherwise returns `null` itself.

Using the scoping functions we described earlier, there is an elegant method to avoid constructs like `if( x != null ) { ... }`. You can instead write the following:

```
x?.run {
  ...
}
```

This does the same but is more expressive; by virtue of the `?.`, the execution of `run{}` happens only if `x` is not `null`.

The `elvis` operator `?:` is also quite useful because it handles cases where you want to calculate an expression only if the receiver variable is `null`, as follows:

```
var x:String? = ...
...
var y:String = x ?: "default"
```

This is the same as `String y = (x != null) ? x : "default");` in Java.

Data Classes

Data classes are classes whose responsibility is to carry structured data. Actually doing something with the data inside the data class usually is not necessary or at least not important.

The declaring of data classes in Kotlin is easy; all you have to do is write the following:

```
data class Person(
       val fName:String,
       val lName:String,
       val age:Int)
```

Or, if you want to use default values for some arguments, use this:

```
data class Person(
       val fName:String="",
       val lName:String,
       val age:Int=0)
```

This simple declaration already defines a constructor, an appropriate `equals()` method for comparison, a default `toString()` implementation, and the ability to be part of a destructuring. To create an object, you just have to write the following:

```
val pers = Person("John","Smith",37)
```

or write a more expressive version, shown here:

```
val pers = Person(fName="John", lName="Smith", age=37)
```

In this case, you can also omit parameters if they have defaults declared.

This and the fact that you can declare classes and functions also inside functions makes it easy to define ad hoc complex function return types, as follows:

```
fun someFun() {
  ...
  data class Person(
       val fName:String,
       val lName:String,
       val age:Int)
  fun innerFun():Person = ...
  ...
  val p1:Person = innerFun()
  val fName1 = p1.fName
  ...
```

Destructuring

A destructuring declaration allows you to multi-assign values or variables. Say you have a data class `Person` as defined in the previous section. You can then write the following:

```
val p:Person = ...
val (fName,lName,age) = p
```

This gives you three different values. The order for data classes is defined by the order of the class's member declaration. Generally, any object that has `component1()`, `component2()`, ... accessors can take part in a destructuring, so you can use destructuring for your own classes as well. This is, for example, by default given for map entries, so you can write the following:

```
val m = mapOf( 1 to "John", 2 to "Greg", ... )
for( (k,v) in m) { ... }
```

Here, the `to` is an infix operator that creates a `Pair` class, which in turn has `fun component1()` and `fun component2()` defined.

As an additional feature to a destructuring declaration, you can use _ wildcards for unused parts, as follows:

```
val p:Person = ...
val (fName,lName,_) = p
```

Multiline String Literals

Multiline string literals in Java always were a little clumsy to define.

```
String s = "First line\n" +
    "Second line";
```

In Kotlin you can define multiline string literals as follows:

```
val s = """
    First line
    Second Line"""
```

You can even get rid of the preceding indent spaces by adding `.trimIndent()` as follows:

```
val s = """
    First line
    Second Line""".trimIndent()
```

This removes the leading newline and the common spaces at the beginning of each line.

Inner Functions and Classes

In Kotlin, functions and classes can also be declared inside other functions, which further helps in structuring your code.

```
fun someFun() {
  ...
  class InnerClass { ... }
  fun innerFun() = ...
  ...
}
```

The scope of such inner constructs is of course limited to the function in which they are declared.

String Interpolation

In Kotlin you can pass values into strings as follows:

```
val i = 7
val s = "And the value of 'i' is ${i}"
```

This is borrowed from the Groovy language, and you can use it for all types since all types have a `toString()` member. The only requirement is that the contents of `${}` evaluate to an expression, so you can even write the following:

```
val i1 = 7
val i2 = 8
val s = "The sum is: ${i1+i2}"
```

or write more complex constructs using method calls and lambda functions:

```
val s = "8 + 1 is: ${ { i: Int -> i + 1 }(8) }"
```

Qualified "this"

If `this` is not what you want but you instead want to refer to `this` from an outer context, in Kotlin you use the `@` qualifier as follows:

```
class A {
    val b = 7
    init {
        val p = arrayOf(8,9).apply {
            this[0] += this@A.b
        }
        ...
    }
}
```

Delegation

Kotlin allows for easily following the delegation pattern. Here's an example:

```
interface Printer {
    fun print()
}

class PrinterImpl(val x: Int) : Printer {
    override fun print() { print(x) }
}

class Derived(b: Printer) : Printer by b
```

Here, the class `Derived` is of type `Printer` and delegates all its method calls to the `b` object. So, you can write the following:

```
val pi = PrinterImpl(7)
Derived(pi).print()
```

You are free to overwrite method calls at will, so you can adapt the delegate to use new functionalities.

```
class Derived(val b: Printer) : Printer by b {
    override fun print() {
         print("Printing:")
         b.print()
    }
}
```

Renamed Imports

In some cases, imported classes might use long names but you use them often, so you wish they would have shorter names. For example, say you often use `SimpleDateFormat` classes in your code and don't want to write the full class name all the time. To help us with that and shorten this a little, you can introduce import aliases and write the following:

```
import java.text.SimpleDateFormat as SDF
```

Henceforth, you can use `SDF` as a substitute for `SimpleDateFormat`, as follows:

```
val dateStr = SDF("yyyy-MM-dd").format(Date())
```

Don't overuse this feature, though, because otherwise your fellow developers need to memorize too many new names, which makes your code hard to read.

Kotlin on JavaScript

If you hear Android and Kotlin together, the obvious thing you will think is that Kotlin serves as a substitute for Java and addresses the Android Runtime and Android APIs. But there is another possibility, which is not that obvious but nevertheless opens interesting possibilities. If you look at Kotlin alone, you will see that it can create bytecode to be run on a Java virtual machine or on a somewhat Java-like Dalvik Virtual Machine in the case of Android. Or it can produce JavaScript to be used in a browser. The question is, can we use that in Android as well? The answer is yes, and in the following sections I will show you how this can be done.

Creating a JavaScript Module

We start with a JavaScript module containing Kotlin files that are compiled to JavaScript files. There is nothing like a JavaScript module wizard available when you start a new module, but we can easily start with a standard smartphone app module and convert it to serve our needs.

In an Android Studio project, select New ➤ New Module and then choose Phone & Tablet Module. Give it a decent name, say `kotlinjsSample` for now. Once the module is generated, remove the following folders and files because we don't need them:

```
src/test
src/androidTest
src/main/java
src/main/res
src/main/AndroidManifest.xml
```

> **Note** If you want to do that removal from inside Android Studio, you have to switch the view type from Android to Project first.

Instead, add two folders.

```
src/main/kotlinjs
src/main/web
```

Now replace the contents of the module's `build.gradle` file to read as follows:

```
buildscript {
  ext.kotlin_version = '1.2.31'
  repositories {
      mavenCentral()
  }
  dependencies {
      classpath "org.jetbrains.kotlin:" +
            "kotlin-gradle-plugin:$kotlin_version"
  }
}

apply plugin: 'kotlin2js'

sourceSets {
  main.kotlin.srcDirs += 'src/main/kotlinjs'
}
task prepareForExport(type: Jar) {
  baseName = project.name + '-all'
  from {
      configurations.compile.collect {
            it.isDirectory() ? it : zipTree(it) } +
      'src/main/web'
  }
  with jar
}

repositories {
  mavenCentral()
}

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

This build file enables the Kotlin ➤ JavaScript compiler and introduces a new export task.

You can now open the Gradle view on the right side of Android Studio's window, and there under `others`, you will find the task `prepareForExport`. To run it, double-click it. After that, inside `build/libs` you will find a new file `kotlinjsSample-all.jar`. It is this file that represents the JavaScript module for use by other apps or modules.

Create the file `Main.kt` inside `src/main/kotlinjs` and add content to it as follows:

```
import kotlin.browser.document

fun main(args: Array<String>) {
    val message = "Hello JavaScript!"
    document.getElementById("cont")!!.innerHTML = message
}
```

In the end we will be targeting a web site, so we need a first HTML page as well. Make it the standard landing page `index.html`, create it inside `src/main/web`, and enter the following:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
    <title>Kotlin-JavaScript</title>
  </head>
  <body>
    <span id="cont"></span>
    <script type="text/javascript"
           src="kotlin.js"></script>
    <script type="text/javascript"
           src="kotlinjsSample.js"></script>
  </body>
</html>
```

Execute the task `prepareForExport` once again to let the module output artifact reflect the changes we just made.

Using the JavaScript Module

To use the JavaScript module we constructed in the previous section, add a couple of lines in the app's `build.gradle` file, as shown here:

```
task syncKotlinJs(type: Copy) {
  from zipTree('../kotlinjsSample/build/libs/' +
               'kotlinjsSample-all.jar')
  into 'src/main/assets/kotlinjs'
}
preBuild.dependsOn(syncKotlinJs)
```

This will import the JavaScript module's output file and extract it inside the `assets` folder of the app. This extra build task gets executed automatically for you during a normal build by virtue of the `dependsOn()` declaration.

Now inside your layout file place a `WebView` element, maybe as follows:

```
<WebView
     android:id="@+id/wv"
     android:layout_width="match_parent"
     android:layout_height="match_parent">
</WebView>
```

To fill that view with a web page inside your main activity's `onCreate()` callback, write the following:

```
wv.webChromeClient = WebChromeClient()
wv.settings.javaScriptEnabled = true
wv.loadUrl("file:///android_asset/kotlinjs/index.html")
```

This will enable JavaScript support for the `WebView` widget and load the main HTML page from the JavaScript module.

As an extension, you might want to connect the JavaScript inside the web page to the Kotlin code from the app (not the JavaScript module). This is not overly complicated; you just have to add the following:

```
class JsObject {
    @JavascriptInterface
    override fun toString(): String {
      return "Hi from injectedObject"
    }
}
wv.addJavascriptInterface(JsObject(), "injectedObject")
```

Henceforth you can use `injectedObject` from the JavaScript module as follows:

```
val message = "Hello JavaScript! injected=" +
      window["injectedObject"]
```

Using these techniques you could design your complete app using HTML, CSS, Kotlin transcribing to JavaScript, and a couple of accessor objects to address Android APIs.

Chapter 11

Building

In this chapter, we talk about the building process of your apps. Although building an app with source files can be done both using a terminal and using the graphical interface of the Android Studio IDE, this is not an introduction to Android Studio nor a code reference. For this type of in-depth instruction, please refer to the help included or to other books and online resources.

What we will do in this chapter is look at build-related concepts and methods for adapting the build process to your needs.

Build-Related Files

Once you create a new project inside Android Studio, you will see the following build-related files:

- **build.gradle**

  This is the top-level project-related build file. It contains the declaration of repositories and dependencies common to all modules the project contains. There is normally no need for you to edit this file for simple apps.

- **gradle.properties**

  This contains technical settings related to Gradle builds. There is normally no need for you to edit this file.

- **gradlew and gradlew.bat**

  These are wrapper scripts so you can run builds using a terminal instead of the Android Studio IDE.

- **local.properties**

  This holds generated technical properties related to your Android Studio installation. You should not edit this file.

P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_11

- **settings.gradle**

  This tells you which modules are part of the project. Android Studio will handle this file if you add new modules.

- **app/build.gradle**

  This is a module-related build file. This is where important dependencies and the build process for the module are configured. Android Studio will create a first module named `app` including the corresponding build file for you, but `app` as a name is just a convention. Additional modules will have different names you choose at will, and they all have their own build files. It is even possible to rename `app` to a different name that better suits your needs, if you like.

Module Configuration

Each module of a project contains its own build file called `build.gradle`. If you let Android Studio create a new project or module for you, it also creates an initial build file for you. Such a basic build file for a module with Kotlin support looks like this (disregard the ¬ and the following line breaks):

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

android {
  compileSdkVersion 26
  defaultConfig {
      applicationId "de.pspaeth.xyz"
      minSdkVersion 16
      targetSdkVersion 26
      versionCode 1
      versionName "1.0"
      testInstrumentationRunner ¬
        "android.support.test.runner.AndroidJUnitRunner"
  }
  buildTypes {
      release {
          minifyEnabled false
          proguardFiles ¬
            getDefaultProguardFile( ¬
              "proguard-android.txt"), ¬
            "proguard-rules.pro"
      }
  }
}

dependencies {
  implementation fileTree(dir: 'libs', ¬
      include: ['*.jar'])
```

```
  implementation ¬
    "org.jetbrains.kotlin:kotlin-stdlib-jre7: ¬
          $kotlin_version"
  implementation ¬
    "com.android.support:appcompat-v7:26.1.0"
  implementation ¬
    "com.android.support.constraint: ¬
          constraint-layout:1.0.2"
  testImplementation "junit:junit:4.12"
  androidTestImplementation ¬
    "com.android.support.test:runner:1.0.1"
  androidTestImplementation ¬
    "com.android.support.test.espresso: ¬
          espresso-core:3.0.1"
}
```

Note that "" strings in Gradle can contain ${} placeholders, while ' ' strings cannot. Other than that, they are interchangeable.

Its elements are as follows:

- The `apply plugin:` lines load and apply Gradle plugins necessary for Android and Kotlin development.
- The `android{ }` element specifies settings for the Android plugin.
- The `dependencies{ }` element describes dependencies of the module. The `implementation` keyword means the dependency is needed both for compiling the module and for running it. The latter implies that the dependency gets included in the APK file. Identifiers like `xyzImplementation` refer to a build type or source set `xyz`. You can see that for the unit tests located at `src/test`, the jUnit libraries get added, while for `src/androidTest` both the test runner and `espresso` get used. If you prefer to build types or product flavor, you can substitute the build type name or product flavor name for `xyz`. If you want to reference a variant, which is a combination of a build type and a product flavor, you additionally must declare it inside a `configurations { }` element. Here's an example:

  ```
  configurations {
    // flavor = "free", type = "debug"
    freeDebugCompile {}
  }
  ```

- For `defaultConfig { }` and `buildTypes { }`, see the following sections.

Other keywords inside the `dependencies {...}` section include the following:

- **implementation**

  We talked about this one. It expresses the dependency is needed both for compiling and for running the app.

- **api**

  This is the same as `implementation`, but in addition it lets the dependency leak through to clients of the app.

- **compile**

  This is an old alias for `api`. Don't use it.

- **compileOnly**

  The dependency is needed for compilation but will not be included in the app. This frequently happens for source-only libraries like source code preprocessors and the like.

- **runtimeOnly**

  The dependency is not needed for compilation but will be included in the app.

Module Common Configuration

The element `defaultConfig { ... }` inside a module's `build.gradle` file specifies configuration settings for a build, independent of the variant chosen (see the next section). The possible setting can be looked up in the Android Gradle DSL reference, but a common setup reads like the following:

```
defaultConfig {

  // Uniquely identifies the package for publishing.
  applicationId 'com.example.myapp'

  // The minimum API level required to run the app.
  minSdkVersion 24

  // The API level used to test the app.
  targetSdkVersion 26

  // The version number of your app.
  versionCode 42

  // A user-friendly version name for your app.
  versionName "2.5"
}
```

Module Build Variants

Build variants correspond to different `.apk` files that are generated by the build process. The number of build variants is given by the following:

```
Number of Build Variants =
     (Number of Build Types) x (Number of Product Flavors)
```

Inside Android Studio, you choose the build variant via Build ➤ Select Build Variant in the menu. In the following sections, we describe what build types and product flavors are.

Build Types

Build types correspond to different stages of the app development. If you start a project, Android Studio will set up two build types for you: *development* and *release*. If you open the module's `build.gradle` file, you can see inside `android { ... }` (disregard the ¬, including the following newlines).

```
buildTypes {
   release {
     minifyEnabled false
     proguardFiles ¬
       getDefaultProguardFile('proguard-android.txt'), ¬
       'proguard-rules.pro'
   }
}
```

Even though you don't see a `debug` type here, it exists. The fact that it doesn't appear just means the `debug` type uses its default settings. If you need to change the defaults, just add a `debug` section as follows:

```
buildTypes {
   release {
     ...
   }
   debug {
     ...
   }
}
```

You are not restricted to use one of the predefined build types. You can define additional build types, as for example here:

```
buildTypes {
   release {
     ...
   }
   debug {
     ...
   }
   integration {
     initWith debug
     manifestPlaceholders = ¬
            [hostName:"internal.mycompany.com"]
     applicationIdSuffix ".integration"
   }
}
```

This defines a new build type called `integration` that inherits from `debug` by virtue of `initWith` and otherwise adds a custom app file suffix and provides a placeholder to be used in the manifest file. The settings you can specify there are rather numerous. You can find them if you enter *android gradle plugin dsl reference* in your favorite search engine.

Another identifier we haven't talked about yet is the `proguardFiles` identifier. That one is used for filtering and/or obfuscating files that are to be included in the app before distributing it. If you use it for filtering, please first weight the benefit against the effort. With modern devices, saving a few megabytes doesn't play a big role nowadays. And if you want to use it for obfuscation, note that this might cause trouble if reflection gets used by either your code or from the libraries referred to. And obfuscation does not really prevent hijackers from using your code after decompilation; it just makes it a little harder. So, carefully consider the advantages of using `proguard`. If you think it will suit your needs, you can find details about how to use it in the online documentation.

Product Flavors

Product flavors allow distinctions between things like different feature sets or different device requirements, but you can draw the distinction wherever best suits you.

By default Android Studio doesn't prepare different product flavors for a new project or module. If you need them, you must add a `productFlavors { ... }` section inside the `android { ... }` element of the file `build.gradle`. Here's an example:

```
buildTypes {...}
flavorDimensions "monetary"
productFlavors {
    free {
        dimension "monetary"
        applicationIdSuffix ".free"
        versionNameSuffix "-free"
    }
    paid {
        dimension "monetary"
        applicationIdSuffix ".paid"
        versionNameSuffix "-paid"
    }
}
```

Here, you can look at the possible settings in the Android Gradle DSL reference. This will lead to APKs of the following form:

```
app-free-debug.apk
app-paid-debug.apk
app-free-release.apk
app-paid-release.apk
```

You can even extend the dimensionality. If you add more elements to the `flavorDimensions` line, for example `flavorDimensions "monetary", "apilevel"`, you can add more flavors.

```
flavorDimensions "monetary", "apilevel"
productFlavors {
    free {
        dimension "monetary" ... }
    paid {
        dimension "monetary" ... }
    sinceapi21 {
        dimension "apilevel"
        versionNameSuffix "-api21" ... }
    sinceapi24 {
        dimension "apilevel"
        versionNameSuffix "-api24" ... }
}
```

This in the end will give you the following set of APK files:

```
app-free-api21-debug.apk
app-paid-api21-debug.apk
app-free-api21-release.apk
app-paid-api21-release.apk
app-free-api24-debug.apk
app-paid-api24-debug.apk
app-free-api24-release.apk
app-paid-api24-release.apk
```

To filter out certain possible variants, add a `variantFilter` element into the build file and write the following:

```
variantFilter { variant ->
  def names = variant.flavors*.name  // this is an array
  // To filter out variants, make a check here and then
  // do a "setIgnore(true)" if you don't need a variant.
  // This is just an example:
  if (names.contains("sinceapi24") &&
        names.contains("free")) {
    setIgnore(true)
  }
}
```

Source Sets

If you create a project in Android Studio and switch to the Project view, you can see that there is a `main` folder inside the `src` folder. This corresponds to the `main` source set, which is the single default source set configured and used by default. See Figure 11-1.

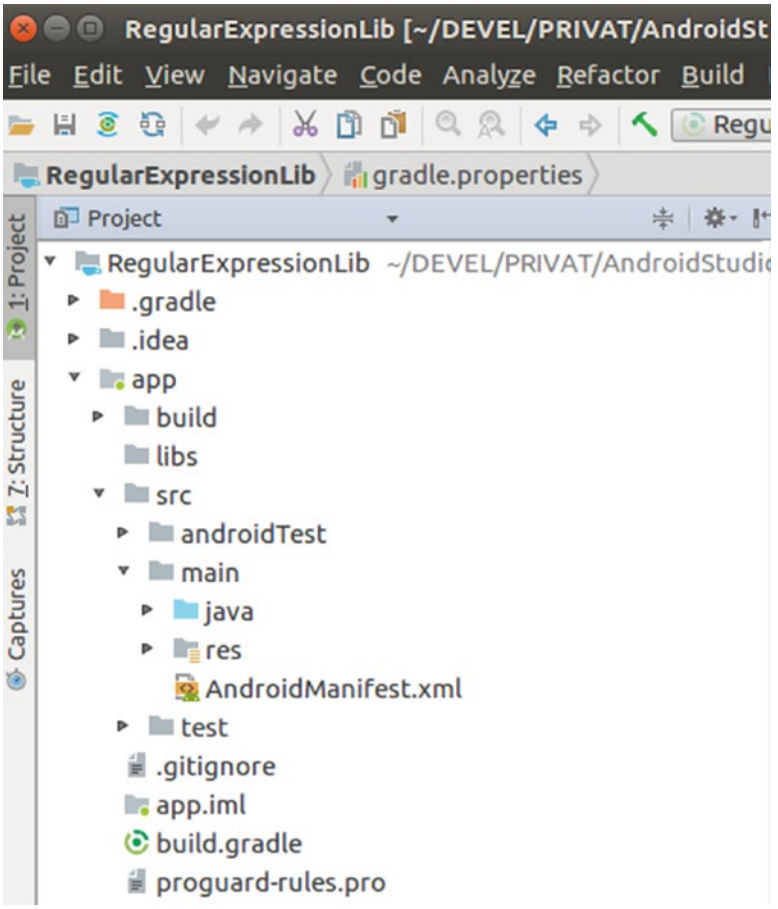

Figure 11-1. *The main source set*

You can have more sets, and they correspond to the build types, the product flavors, and the build variants. As soon as you add more source sets, a build will lead to merging the current build variant, the build type it includes, the product flavor it includes, and finally the main source set. To see which source sets will be included in a build, open the Gradle view on the right side of the window, and run the `sourceSets` task. This will produce a long listing, and you can see entries like the following:

```
main
Java sources: [app/src/main/java]

debug
Java sources: [app/src/debug/java]

free
Java sources: [app/src/free/java]

freeSinceapi21
Java sources: [app/src/freeSinceapi21/java]
```

```
freeSinceapi21Debug
Java sources: [app/src/freeSinceapi21Debug/java]

freeSinceapi21Release
Java sources: [app/src/freeSinceapi21Release/java]

paid
Java sources: [app/src/paid/java]

paidSinceapi21
Java sources: [app/src/paidSinceapi21/java]

release
Java sources: [app/src/release/java]

sinceapi21
Java sources: [app/src/sinceapi21/java]
```

This will tell you that if you choose a build variant called `freeSinceapi21Debug`, the build process will look into these folders for classes:

```
app/src/freeSinceapi21Debug/java
app/src/freeSinceapi21/java
app/src/free/java
app/src/sinceapi21/java
app/src/debug/java
app/src/main/java
```

Likewise, it will look into the corresponding folders for resources, assets, and the `AndroidManifest.xml` file. While the Java or Kotlin classes must not repeat in such a build chain, the manifest files and resource and assets files will be merged by the build process.

Inside the `dependencies { ... }` section of file `build.gradle`, you can dispatch dependencies according to build variants. Just add a camel-cased version of the source set in front of any of the settings there. For example, if for the `freeSinceapi21` variant you want to include a compile dependency of `:mylib`, write the following:

```
freeSinceapi21Compile ':mylib'
```

Running a Build from the Console

You don't have to use Android Studio to build apps. While it is a good idea to bootstrap an app project using Android Studio, after this you can build apps using a terminal. This is what the Gradle wrapper scripts `gradlew` and `gradlew.bat` are for. The first one is for Linux, and the second one is for Windows. In the following paragraphs, we will take a look at some command-line commands for Linux; if you have a Windows development machine, just use the `BAT` script instead.

In the preceding sections, we have seen that the basic building blocks of each build consist of one or more tasks that get executed during the build. So, we first want to know which tasks actually exist. For this aim, to list all the tasks available, enter the following:

```
./gradlew tasks
```

This will give you an extensive list and some description of each task. In the following sections, we will take a look at some of these tasks.

To build the app APK file for build type `debug` or `release`, enter one of the following:

```
./gradlew assembleDebug
./gradlew assembleRelease
```

This creates an APK file inside `<PROJECT>/<MODULE>/build/outputs`. Of course, you can also specify any custom build type you defined inside `build.gradle`.

To build the debug type APK and then install it on a connected device or emulator, enter the following:

```
./gradlew installDebug
```

Here, for the `Debug` part in the argument, you can substitute any build variant using the variant's camel-cased name. This installs the app on connected devices. It does not automatically run it, though; you have to do that manually! To install *and* run an app, please see Chapter 18.

If you want to find out which dependencies any of your app's module has, see the dependency tree and enter the following or with `app` substituted out for the module name in question:

```
./gradlew dependencies :app:dependencies
```

This provides a rather lengthy listing, so you might want to pipe it into a file and then investigate the result in an editor.

```
./gradlew dependencies :app:dependencies > deps.txt
```

Signing

Each app's APK file needs to be signed before it can be run on a device. For the `debug` build type, a suitable signing configuration will be chosen for you automatically, so for the debugging development stage, you don't need to care about signing.

A release APK, however, needs a proper signing configuration. If you use Android Studio's Build ➤ Generate Signed APK menu item, Android Studio will help you create and/or use an appropriate key. But you can also specify the signing configuration inside the module's `build.gradle` file. To do so, add a `signingConfigs { ... }` section as follows:

```
android {
  ...
  defaultConfig {...}
  signingConfigs {
      release {
          storeFile file("myrelease.keystore")
           storePassword "passwd"
          keyAlias "MyReleaseKey"
          keyPassword "passwd"
      }
  }
  buildTypes {
      release {
          ...
          signingConfig signingConfigs.release
      }
  }
}
```

Also, from inside the release build type, refer to a signing config as shown at `signingConfig` inside the listing. The keystore you need to provide for that is a standard Java keystore; please see Java's online documentation to learn how to build one. Or, you can let Android Studio help you create a keystore using the dialog that pops up when you choose Build ➤ Generate Signed APK in the menu.

Chapter 12

Communication

Communication is about sending data through component or app or device boundaries. A standardized way for the components of one or more apps to communicate with each other is by using *broadcasts*, which were discussed in Chapter 5.

Another possibility for inter-app communication on one device is to use `ResultReceiver` objects, which are passed by intents. Despite their name, they can be used to send data back to an invoker not only when an invoked component has done its work but also anytime while it is alive. We used them at a couple of places in this book, but in this chapter we will revise using how we use them to have all communication means together.

For communication through device boundaries, the options are numerous, which is even more true if we can use a cloud-based communication platform. We will be talking about inter-app communication using both cloud-based services and direct communication over the internet.

ResultReceiver Classes

A `ResultReceiver` object can be passed from any one component to another component by assigning it to an intent, so you can use it to send data between components of any kind, provided they live on the same device.

We first subclass a `ResultReceiver` that will later receive messages from an invoked component and write the following:

```
class MyResultReceiver : ResultReceiver(null) {
  companion object {
    val INTENT_KEY = "my.result.receiver"
    val DATA_KEY = "data.key"
  }
```


P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_12

```
  override fun onReceiveResult(resultCode: Int,
        resultData: Bundle?) {
    super.onReceiveResult(resultCode, resultData)
    val d = resultData?.get(DATA_KEY) as String
    Log.e("LOG", "Received: " + d)
  }
}
```

Of course, you can write more meaningful things inside its `onReceiveResult()` function.

To pass an instance of `MyResultReceiver` to an invoked component, we can now write the following or any other means to invoke another component:

```
Intent(this, CalledActivity::class.java).apply {
    putExtra(MyResultReceiver.INTENT_KEY,
          MyResultReceiver())
}.run{ startActivity(this) }
```

Inside the invoked component, you can now at any suitable place send data to the invoking component via something like this:

```
var myReceiver:ResultReceiver? = null

override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_called)
  ...
  myReceiver = intent.
        getParcelableExtra<ResultReceiver>(
        MyResultReceiver.INTENT_KEY)
}

fun go(v: View) {
  val bndl = Bundle().apply {
      putString(MyResultReceiver.DATA_KEY,
        "Hello from called component")
  }
  myReceiver?.send(42, bndl) ?:
        throw IllegalStateException("myReceiver is null")
}
```

Inside a production environment, you additionally need to take care of checking whether the recipient is still alive. I left this check out for brevity. Also note that on the sending side a reference to the `ResultReceiver` implementation is actually not needed; if you communicate through app boundaries, you can just write the following:

```
...
val INTENT_KEY = "my.result.receiver"
val DATA_KEY = "data.key"
...
val myReceiver = intent.
    getParcelableExtra<ResultReceiver>(
    INTENT_KEY)
```

```
...
val bndl = Bundle().apply {
  putString(DATA_KEY,
    "Hello from called component")
}
myReceiver?.send(42, bndl)
```

Firebase Cloud Messaging

Firebase Cloud Messaging (FCM) is a cloud-based message broker you can use to send and receive messages from various devices, including other operating systems like Apple iOS. The idea is as follows: you register an app in the Firebase console and henceforth can receive and send messages in connected devices, including other installations of your app on other devices.

> **Note** Firebase Cloud Messaging is a successor of Google Cloud Messaging (GCM). The documentation says you should favor FCM over GCM. In this book we talk about FCM; if you need information about GCM, please consult the online resources.

To start FCM from inside Android Studio, from your open project go to Tools ➤ Firebase for various wizards. Select Cloud Messaging and then Set up Firebase Cloud Messaging. If you follow the instructions there, you will end up using two services.

A subclass of `FirebaseInstanceIdService` where you will receive a message token. The class basically looks like this:

```
class MyFirebaseInstanceIdService :
      FirebaseInstanceIdService() {
  override
  fun onTokenRefresh() {
      // Get updated InstanceID token.
      val refreshedToken =
          FirebaseInstanceId.getInstance().token
      Log.d(TAG, "Refreshed token: " +
          refreshedToken!!)
  }
}
```

It has a corresponding entry inside `AndroidManifest.xml`.

```
<service
  android:name=".MyFirebaseInstanceIdService"
  android:enabled="true"
  android:exported="true">
    <intent-filter>
      <action android:name=
          "com.google.firebase.INSTANCE_ID_EVENT"/>
    </intent-filter>
</service>
```

The token you receive here when you first start an app that is connected to Firebase is important; you need it to use the Firebase-based communication channel. It is subject to infrequent automated renewal, so you need to find a way to reliably store the token whenever you receive it in this service. Do yourself a favor: unless you implemented a way to store the token, be sure to save the token you receive in the logs because recovering a lost token results in annoying administrative work.

Another service signs responsible for receiving FCM-based messages. It could read as follows:

```
class MyFirebaseMessagingService :
      FirebaseMessagingService() {
  override
  fun onMessageReceived(remoteMessage:
        RemoteMessage) {
    // ...
    // Check if message contains a data payload.
    if (remoteMessage.data.size > 0) {
      Log.d(TAG, "Message data payload: " +
            remoteMessage.data)

    // Implement a logic:
    // For long-running tasks (10 seconds or more)
    // use Firebase Job Dispatcher.
    scheduleJob()
    // ...or else handle message within 10 seconds
    // handleNow()
  }

  // Message contains a notification payload?
    remoteMessage.notification?.run {
      Log.d(TAG, "Message Notification Body: " +
            body)
    }
  }

  private fun handleNow() {
      Log.e("LOG","handleNow()")
  }

  private fun scheduleJob() {
      Log.e("LOG","scheduleJob()")
  }
}
```

This too has a corresponding entry in `AndroidManifest.xml`.

```
<service
    android:name=".MyFirebaseMessagingService"
    android:enabled="true"
    android:exported="true">
  <intent-filter>
    <action android:name=
          "com.google.firebase.MESSAGING_EVENT"/>
  </intent-filter>
</service>
```

For this all to work, you need to have Firebase active in your Google account. There are several options, and for a high-traffic messaging service, you need to buy a plan. The free variant (as of March 2018), however, will give you more than enough power for development and tests.

If all is set up correctly, you can use the web-based Firebase console to test sending messages to your running app and see the message arriving there in the logs.

> **Note** Firebase is more than just messaging; please consult the online documentation and information you find in the Firebase console, as well as the information Android Studio gives you, to learn what else can be done.

For sending messages, the suggested solution is to set up a trusted environment in the form of an application server. This is beyond the scope of the book, but the online Firebase documentation gives you various hints to get started with that matter.

Communication with Backends

Using a cloud-based provider like Firebase for connecting your app to other apps on other devices, as described in the preceding section, certainly exhibits different merits. You have a reliable message broker with message backup facilities, analytics, and more.

But using the cloud has its disadvantages as well. Your data, whether it's encrypted or not, will leave your house even for corporate apps, and you cannot be 100 percent sure the provider will not change the API at some point in the future, forcing you to change your app. So, if you need more control, you can abandon the cloud and use direct networking instead.

For directly using network protocols to communicate with devices or application servers, you basically have two options.

- **Use javax.net.ssl.HttpsURLConnection**

  This provides for a low-level connectivity, but with TLS, streaming capabilities, timeouts, and connection pooling included. As you can see from the class name, it is part of the standard Java API, so you will find lots of information about it on the Web. We nevertheless give a description in the following section.

- **Use the Volley API included with Android**

  This is a higher-level wrapper around basic networking functions. Using Volley considerably simplifies network-based development, so it is generally the first candidate for using networking in Android.

In both cases you need to add appropriate permissions inside `AndroidManifest.xml`.

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

Communication with HttpsURLConnection

Before using a network communication API, we need to make sure networking operations happen in the background; modern Android versions even won't allow you to perform networking in the UI thread. But even without that restriction, it is highly recommended to always perform networking in a background task. We talked about background operation in Chapter 9. A first method you want to look at is running network operations inside an `AsyncTask`, but you are free to choose other means as well. The following sections assume the snippets presented there are running in the background.

Using class `HttpsURLConnection`-based communication boils down to the following:

```
fun convertStreamToString(istr: InputStream): String {
  val s = Scanner(istr).useDelimiter("\\A")
  return if (s.hasNext()) s.next() else ""
}

// This is a convention for emulated devices
// addressing the host (development PC)
val HOST_IP = "10.0.2.2"

val url = "https://${HOST_IP}:6699/test/person"
var stream: InputStream? = null
var connection: HttpsURLConnection? = null
var result: String? = null
try {
  connection = (URL(uri.toString()).openConnection()
        as HttpsURLConnection).apply {

    // ! ONLY FOR TESTING ! No SSL hostname verification
    class TrustAllHostNameVerifier : HostnameVerifier {
      override
      fun verify(hostname: String, session: SSLSession):
            Boolean = true
    }
    hostnameVerifier = TrustAllHostNameVerifier()

    // Timeout for reading InputStream set to 3000ms
    readTimeout = 3000
    // Timeout for connect() set to 3000ms.
    connectTimeout = 3000
    // For this use case, set HTTP method to GET.
    requestMethod = "GET"
    // Already true by default, just telling. Needs to
    // be true since this request is carrying an input
    // (response) body.
    doInput = true
    // Open communication link
    connect()
    responseCode.takeIf {
          it != HttpsURLConnection.HTTP_OK }?.run {
      throw IOException("HTTP error code: $this")
    }
```

```
    // Retrieve the response body
    stream = inputStream?.also {
      result = it.let { convertStreamToString(it) }
    }
  }
} finally {
  stream?.close()
  connection?.disconnect()
}

Log.e("LOG", result)
```

This example tries to access a GET URL of `https://10.0.2.2:6699/test/person` that targets your development PC and prints the result on the logs.

Note that if your server happens to hold a self-signed certificate for SSL, you must at an initialization place, say inside the `onCreate()` callback, add the following:

```
val trustAllCerts =
      arrayOf<TrustManager>(object : X509TrustManager {
    override
    fun getAcceptedIssuers():
      Array<java.security.cert.X509Certificate>? = null
    override
    fun checkClientTrusted(
      certs: Array<java.security.cert.X509Certificate>,
      authType: String) {
    }
    override
    fun checkServerTrusted(
      certs: Array<java.security.cert.X509Certificate>,
      authType: String) {
    }
})

SSLContext.getInstance("SSL").apply {
  init(null, trustAllCerts, java.security.SecureRandom())
}.apply {
  HttpsURLConnection.setDefaultSSLSocketFactory(
        socketFactory)
}
```

Otherwise, the previous code will complain and fail. Of course, you favor officially signed certificates over self-signed certificates in production code.

Networking with Volley

Volley is a networking library that simplifies networking for Android. First, Volley sends its work to the background by itself; you don't have to take care of that. Other goodies provided by Volley are the following:

- Scheduling mechanisms
- Parallel working several requests
- Handling of JSON requests and responses
- Caching
- Diagnosis tools

To start developing with Volley, add the dependency to your module's `build.gradle` file, as shown here:

```
dependencies {
  ...
  implementation 'com.android.volley:volley:1.1.0'
}
```

The next thing to do is set up a `RequestQueue` that Volley uses to handle requests in the background. The easiest way to do that is to write the following in an activity:

```
val queue = Volley.newRequestQueue(this)
```

But you can also customize the creation of a `RequestQueue` and instead write the following:

```
val CACHE_CAPACITY = 1024 * 1024 // 1MB
val cache = DiskBasedCache(cacheDir, CACHE_CAPACITY)
// ... or a different implementation

val network = BasicNetwork(HurlStack())
// ... or a different implementation

val requestQueue = RequestQueue(cache, network).apply {
  start()
}
```

The question is, under which scope is the request queue best defined? We could create and run the request queue in an activity's scope, which means that the queue needs to be re-created each time the activity gets re-created itself. This is a valid option, but the documentation suggests using the application scope instead to reduce the re-creation of caches. The recommended way is to use the `Singleton` pattern, which results in the following:

```
class RequestQueueSingleton
      constructor (context: Context) {
  companion object {
    @Volatile
    private var INSTANCE: RequestQueueSingleton? = null
    fun getInstance(context: Context) =
```

```
        INSTANCE ?: synchronized(this) {
            INSTANCE ?: RequestQueueSingleton(context)
        }
  }
  val requestQueue: RequestQueue by lazy {
    val alwaysTrusting = object : HurlStack() {
      override
      fun createConnection(url: URL): HttpURLConnection {
        fun getHostnameVerifier():HostnameVerifier {
          return object : HostnameVerifier {
            override
            fun verify(hostname:String,
                  session:SSLSession):Boolean = true
          }
        }
        return (super.createConnection(url) as
              HttpsURLConnection).apply {
          hostnameVerifier = getHostnameVerifier()
        }
      }
  }
  // Using the Application context is important.
  // This is for testing:
  Volley.newRequestQueue(context.applicationContext,
          alwaysTrusting)
    // ... for production use:
    // Volley.newRequestQueue(context.applicationContext)
  }
}
```

For development and testing purposes, an accept-all SSL hostname verifier was added.

So, instead of writing `val queue = Volley.newRequestQueue(this)` or `val requestQueue = RequestQueue(...)` as shown earlier, you then use the following:

```
val queue = RequestQueueSingleton(this).requestQueue
```

Now for sending a string request, you have to write the following:

```
// This is a convention for emulated devices
// addressing the host (development PC)
val HOST_IP = "10.0.2.2"

val stringRequest =
    StringRequest(Request.Method.GET,
          "https://${HOST_IP}:6699/test/person",
          Response.Listener<String> { response ->
            val shortened =
                response.substring(0,
                      Math.min(response.length, 500))
            tv.text = "Response is: ${shortened}"
          },
```

```
          Response.ErrorListener { err ->
            Log.e("LOG", err.toString())
            tv.text = "That didn't work!"
          })
queue.add(stringRequest)
```

Here, `tv` points to a `TextView` UI element. For that to work, you need to have a server responding to `https://localhost:6699/test/person`. Note that the response listener automatically runs on the UI thread, so you don't have to take care of that yourself.

To cancel single requests, use `cancel()` on the request object anywhere. You can also cancel a group of requests. Add a tag to each request in question as in `val stringRequest = ... .apply {tag = "TheTag"}` and then write `queue?.cancelAll( "TheTag" )`. Volley makes sure the response listener never gets called once a request is canceled.

To request a JSON object or JSON array, you have to substitute the following:

```
val request =
    JsonArrayRequest(Request.Method.GET, ...)
```

or the following for the `StringRequest` we used previously:

```
val request =
    JsonObjectRequest(Request.Method.GET, ...)
```

For example, for a JSON request and the POST method, you can write the following:

```
val reqObj:JSONObject =
      JSONObject("""{"a":7, "b":"Hello"}""")
val json1 = JsonObjectRequest(Request.Method.POST,
      "https://${HOST_IP}:6699/test/json",
      reqObj,
      Response.Listener<JSONObject> { response ->
          Log.e("LOG", "Response: ${response}")
      },
      Response.ErrorListener{ err ->
          Log.e("LOG", "Error: ${err}")
      })
```

Volley can do more for you; you can use other HTTP methods like `PUT` and also write custom request handling and return other data types. Please see Volley's online documentation or its API documentation for more details.

Setting Up a Test Server

This is not really an Android topic and not even anything that necessarily has to do with Kotlin, but to test the communication, you need to have some kind of web server running. To make things easy, I usually configure a simple yet powerful server based on Groovy and Spark (not Apache Spark but instead Java Spark from http://sparkjava.com/).

To use it in Eclipse, first install the Groovy plugin. Then create a Maven project and add the dependencies as follows:

```
<dependency>
  <groupId>com.sparkjava</groupId>
  <artifactId>spark-core</artifactId>
  <version>2.7.2</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.25</version>
  <scope>test</scope>
</dependency>
```

After that, create a Java keystore file, write a Groovy script, and start it.

```
import static spark.Spark.*

def keystoreFilePath = "keystore.jks"
def keystorePassword = "passw7%d"
def truststoreFilePath = null
def truststorePassword = null

secure(keystoreFilePath, keystorePassword,
      truststoreFilePath, truststorePassword)
port(6699)

get("/test/person", { req, res -> "Hello World" })

post("/test/json", { req, res ->
    println(req.body())
    '{ "msg":"Hello World", "val":7 }'
})
```

> **Caution** To avoid Servlet API version clashes, remove the dependency on the Servlet API in the Groovy settings dialog that you open by right-clicking Groovy Libraries in the project and selecting Properties.

To create a keystore file under Linux, you could use the Bash script like that following, with the Java path adapted:

```
#!/bin/bash
export JAVA_HOME=/opt/jdk
$JAVA_HOME/bin/keytool -genkey -keyalg RSA \
  -alias selfsigned -keystore keystore.jks \
  -storepass passw7%d -validity 360 -keysize 2048
```

Android and NFC

NFC is for short-range wireless connectivity for the transport of small data packages between NFC-capable devices. The range is limited to a few centimeters between the communication partners. These are typical use cases:

- Connecting and then reading from or writing to NFC tags
- Connecting and then communicating with other NFC-capable devices (peer-to-peer mode)
- Emulating an NFC card by connecting and then communicating with NFC card readers and writers

To start developing an app that speaks NFC, you need to acquire the permission to do so inside `AndroidManifest.xml`.

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

To also limit visibility in the Google Play store, add the following to the same file:

```
<uses-feature android:name="android.hardware.nfc"
      android:required="true" />
```

Talking to NFC Tags

Once a device with NFC enabled discovers an NFC tag in the vicinity, it tries to dispatch the tag according to a certain algorithm. If the systems determines an NDEF datum and finds an intent filter that is able to handle NDEF, the corresponding component gets called. If the tag does not exhibit NDEF data but otherwise identifies itself by providing information about technology and/or payload, this set of data gets mapped to a "tech" record, and the system tries to find a component that is able to handle it. If both fail, the discovery information is limited to the fact that an NFC tag was discovered. In this case, the system tries to find a component that can handle NFC tags without NDEF and without the "tech" type data.

Based on the information found on the NFC tag, Android also creates a URI and a MIME type you can use for intent filters. The procedure for that is described in more detail on the page "NFC Basics" of the Android online developer documentation; enter *android develop nfc basics* in your favorite search engine to find it.

For writing appropriate intent filters, please see Chapter 3, with the addition that for "tech" style discovery you need to add a certain `<meta-data>` element inside `<activity>` as follows:

```
<meta-data android:name="android.nfc.action.
 TECH_DISCOVERED"
  android:resource="@xml/nfc_tech_filter" />
```

This points to a file called `nfc_tech_filter.xml` inside `res/xml`, containing the following or any subset of it:

```
<resources xmlns:xliff=
      "urn:oasis:names:tc:xliff:document:1.2">
  <tech-list>
      <tech>android.nfc.tech.IsoDep</tech>
      <tech>android.nfc.tech.NfcA</tech>
      <tech>android.nfc.tech.NfcB</tech>
      <tech>android.nfc.tech.NfcF</tech>
      <tech>android.nfc.tech.NfcV</tech>
      <tech>android.nfc.tech.Ndef</tech>
      <tech>android.nfc.tech.NdefFormatable</tech>
      <tech>android.nfc.tech.MifareClassic</tech>
      <tech>android.nfc.tech.MifareUltralight</tech>
  </tech-list>
</resources>
```

The actions you need to add to the intent filter to contribute to the NFC dispatching process are as follows:

- For NDEF discovery style, use the following:

```
<intent-filter>
  <action android:name=
        "android.nfc.action.NDEF_DISCOVERED"/>
  ...more filter specs...
</intent-filter>
```

- For tech discovery style, use the following:

```
<intent-filter>
  <action android:name=
        "android.nfc.action.TECH_DISCOVERED"/>
</intent-filter>
<meta-data android:name=
      "android.nfc.action.TECH_DISCOVERED"
      android:resource="@xml/nfc_tech_filter" />
```

- For failback discovery style, use the following:

```
<intent-filter>
  <action android:name=
        "android.nfc.action.TAG_DISCOVERED"/>
  ...more filter specs...
</intent-filter>
```

Once the NFC-related intent gets dispatched, a matching activity can extract NFC information from the intent. To do so, fetch intent extra data via one or a combination of the following:

- `NfcAdapter.EXTRA_TAG`. Required; gives back an `android.nfc.Tag` object.
- `NfcAdapter.EXTRA_NDEF_MESSAGES`. Optional; NDEF messages from the tag. You can retrieve them via the following:

```
        val rawMessages : Parcelable[] =
            intent.getParcelableArrayExtra(
            NfcAdapter.EXTRA_NDEF_MESSAGES)
```

- `NfcAdapter.EXTRA_ID`. Optional; the low-level ID of the tag.

If you want to write to NFC tags, the procedure for that is described on the page "NFC Basics" of the Android online developer documentation.

Peer-to-Peer NFC Data Exchange

Android allows for the NFC communication between two Android devices via its *Beam* technology. The procedure goes as follows: let the activity of an NFC-capable device extend `CreateNdefMessageCallback` and implement the method `createNdefMessage( event : NfcEvent ) : NdefMessage`. Inside this method, create and return an `NdefMessage` as follows:

```
val text = "A NFC message at " +
           System.currentTimeMillis().toString()
val msg = NdefMessage( arrayOf(
        NdefRecord.createMime(
             "application/vnd.com.example.android.beam",
             text.toByteArray() )
) )

/*
 * When a device receives an NFC message with an Android
 * Application Record (AAR) added, the application
 * specified in the AAR is guaranteed to run. The AAR
 * thus overrides the tag dispatch system.
 */
//val msg = NdefMessage( arrayOf(
//      NdefRecord.createMime(
//           "application/vnd.com.example.android.beam",
//           text.toByteArray() ),
//      NdefRecord.createApplicationRecord(
//           "com.example.android.beam")
//) )
return msg
```

An NFC data-receiving app could then in its `onResume()` callback detect whether it got initiated by an NFC discovery action.

```
override
fun onResume() {
    super.onResume()
    // Check to see that the Activity started due to an
    // Android Beam event
    if (NfcAdapter.ACTION_NDEF_DISCOVERED ==
          intent.action) {
        processIntent(intent)
    }
}
```

NFC Card Emulation

Letting an Android device act as if it was a smartcard with an NFC chip requires involved setting and programming tasks. This especially makes sense if you think about security; some Android devices may contain a *secure element* that performs the communication with the card reader on a hardware basis. Some other device may apply *host-based card emulation* to let the device CPU perform the communication. An exhaustive description of all the details for NFC card emulation is beyond the scope of this book, but you can find information on the Web if you open the page "Host-based Card Emulation" in the online developer guides of Android.

That said, we describe the basic artifacts to start with a *host-based card emulation*. The example is based on the HCE example provided by the developer guides of Android, but it's converted to Kotlin and boiled down to important NFC-related aspects only (the example runs under an Apache license; see www.apache.org/licenses/LICENSE-2.0). The code reads as follows:

```
/**
 * This is a sample APDU Service which demonstrates how
 * to interface with the card emulation support added
 * in Android 4.4, KitKat.
 *
 * This sample replies to any requests sent with the
 * string "Hello World". In real-world situations, you
 * would need to modify this code to implement your
 * desired communication protocol.
 *
 * This sample will be invoked for any terminals
 * selecting AIDs of 0xF11111111, 0xF22222222, or
 * 0xF33333333. See src/main/res/xml/aid:list.xml for
 * more details.
 *
 * Note: This is a low-level interface. Unlike the
 * NdefMessage many developers are familiar with for
 * implementing Android Beam in apps, card emulation
 * only provides a byte-array based communication
 * channel. It is left to developers to implement
 * higher level protocol support as needed.
 */
class CardService : HostApduService() {
```

The `onDeactivated()` callback gets called if the connection to the NFC card is lost to let the application know the cause for the disconnection (either a lost link or another AID being selected by the reader).

```
/**
 * Called if the connection to the NFC card is lost.
 * @param reason Either DEACTIVATION_LINK_LOSS or
 *     DEACTIVATION_DESELECTED
 */
override fun onDeactivated(reason: Int) {}
```

The `processCommandApdu()` method will be called when a command APDU has been received. A response APDU can be provided directly by returning a byte-array in this method. In general, response APDUs must be sent as quickly as possible, given that the user is likely holding a device over an NFC reader when this method is called. If there are multiple services that have registered for the same AIDs in their metadata entry, you will get called only if the user has explicitly selected your service, either as a default or just for the next tap. This method is running on the main thread of your application. If you cannot return a response APDU immediately, return `null` and use the `sendResponseApdu()` method later.

```
/**
 * This method will be called when a command APDU has
 * been received from a remote device.
 *
 * @param commandApdu The APDU that received from the
 *     remote device
 * @param extras A bundle containing extra data. May
 *     be null.
 * @return a byte-array containing the response APDU,
 *     or null if no response APDU can be sent
 *     at this point.
*/
override
fun processCommandApdu(commandApdu: ByteArray,
      extras: Bundle): ByteArray {
    Log.i(TAG, "Received APDU: " +
          byteArrayToHexString(commandApdu))
    // If the APDU matches the SELECT AID command for
    // this service, send the loyalty card account
    // number, followed by a SELECT_OK status trailer
    // (0x9000).
    if (Arrays.equals(SELECT_APDU, commandApdu)) {
        val account = AccountStorage.getAccount(this)
        val accountBytes = account!!.toByteArray()
        Log.i(TAG, "Sending account number: $account")
        return concatArrays(accountBytes, SELECT_OK_SW)
    } else {
    return UNKNOWN_CMD_SW
  }
}
```

The companion object contains a couple of constants and utility functions.

```
companion object {
    private val TAG = "CardService"
    // AID for our loyalty card service.
    private val SAMPLE_LOYALTY_CARD_AID = "F222222222"
    // ISO-DEP command HEADER for selecting an AID.
    // Format: [Class | Instruction | Parameter 1 |
    //          Parameter 2]
    private val SELECT_APDU_HEADER = "00A40400"
    // "OK" status word sent in response to SELECT AID
    // command (0x9000)
```

```
    private val SELECT_OK_SW =
          hexStringToByteArray("9000")
    // "UNKNOWN" status word sent in response to
    // invalid APDU command (0x0000)
    private val UNKNOWN_CMD_SW =
          hexStringToByteArray("0000")
    private val SELECT_APDU =
          buildSelectApdu(SAMPLE_LOYALTY_CARD_AID)

/**
 * Build APDU for SELECT AID command. This command
 * indicates which service a reader is
 * interested in communicating with. See
 * ISO 7816-4.
 *
 * @param aid Application ID (AID) to select
 * @return APDU for SELECT AID command
 */
fun buildSelectApdu(aid: String): ByteArray {
    // Format: [CLASS | INSTRUCTION |
    //          PARAMETER 1 | PARAMETER 2 |
    //          LENGTH | DATA]
    return hexStringToByteArray(
          SELECT_APDU_HEADER +
          String.format("%02X",
                aid.length / 2) +
                aid)
          }

          /**
           * Utility method to convert a byte array to a
           * hexadecimal string.
           */
          fun byteArrayToHexString(bytes: ByteArray):
                String {
              val hexArray = charArrayOf('0', '1', '2', '3',
                    '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F')
              val hexChars = CharArray(bytes.size * 2)
              var v: Int
              for (j in bytes.indices) {
                  v = bytes[j].toInt() and 0xFF
                  // Cast bytes[j] to int, treating as
                  // unsigned value
                  hexChars[j * 2] = hexArray[v.ushr(4)]
                  // Select hex character from upper nibble
                  hexChars[j * 2 + 1] = hexArray[v and 0x0F]
                  // Select hex character from lower nibble
              }
              return String(hexChars)
          }
```

```
        /**
         * Utility method to convert a hexadecimal string
         * to a byte string.
         *
         * Behavior with input strings containing
         * non-hexadecimal characters is undefined.
         */
        fun hexStringToByteArray(s: String): ByteArray {
            val len = s.length
            if (len % 2 == 1) {
                // TODO, throw exception
            }
            val data = ByteArray(len / 2)
            var i = 0
            while (i < len) {
                 // Convert each character into a integer
                 // (base-16), then bit-shift into place
            data[i / 2] =
                ((Character.digit(s[i], 16) shl 4) +
                Character.digit(s[i + 1], 16)).
                toByte()
            i += 2
        }
        return data
    }

    /**
     * Utility method to concatenate two byte arrays.
     */
    fun concatArrays(first: ByteArray,
          vararg rest: ByteArray): ByteArray {
        var totalLength = first.size
        for (array in rest) {
            totalLength += array.size
        }
        val result =
              Arrays.copyOf(first, totalLength)
        var offset = first.size
        for (array in rest) {
            System.arraycopy(array, 0,
                  result, offset, array.size)
            offset += array.size
        }
        return result
    }
  }
}
```

The corresponding service declaration inside `AndroidManifest.xml` reads as follows:

```
<service android:name=".CardService"
         android:exported="true"
         android:permission=
               "android.permission.BIND_NFC_SERVICE">
```

```
    <!-- Intent filter indicating that we support
         card emulation. -->
    <intent-filter>
        <action android:name=
              "android.nfc.cardemulation.action.
              HOST_APDU_SERVICE"/>
    <category android:name=
              "android.intent.category.DEFAULT"/>
    </intent-filter>
    <!-- Required XML configuration file, listing the
         AIDs that we are emulating cards
         for. This defines what protocols our card
         emulation service supports. -->
    <meta-data android:name=
         "android.nfc.cardemulation.host_apdu_service"
         android:resource="@xml/aid:list"/>
</service>
```

And we need a file called `aid:list.xml` inside `res/xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- This file defines which AIDs this application
   should emulate cards for.

   Vendor-specific AIDs should always start with an "F",
   according to the ISO 7816 spec. We recommended
   vendor-specific AIDs be at least 6 characters long,
   to provide sufficient uniqueness. Note, however, that
   longer AIDs may impose a burden on non-Android NFC
   terminals. AIDs may not exceed 32 characters
    (16 bytes).

   Additionally, AIDs must always contain an even number
   of characters, in hexadecimal format.

   In order to avoid prompting the user to select which
   service they want to use when the device is scanned,
   this app must be selected as the default handler for
   an AID group by the user, or the terminal must
   select *all* AIDs defined in the category
   simultaneously ("exact match").
-->
<host-apdu-service
      xmlns:android=
          "http://schemas.android.com/apk/res/android"
      android:description="@string/service_name"
      android:requireDeviceUnlock="false">
  <!--
  If category="payment" is used for any aid-groups, you
  must also add an android:apduServiceBanner attribute
  above, like so:
  android:apduServiceBanner="@drawable/settings_banner"

   apduServiceBanner should be 260x96 dp. In pixels,
   that works out to...
```

```
    -  drawable-xxhdpi: 780x288 px
    -  drawable-xhdpi:  520x192 px
    -  drawable-hdpi:   390x144 px
    -  drawable-mdpi:   260x96  px

  The apduServiceBanner is displayed in the "Tap & Pay"
  menu in the system Settings app, and is only displayed
  for apps which implement the "payment" AID category.

  Since this sample is implementing a non-standard card
  type (a loyalty card, specifically), we do not need
  to define a banner.

  Important: category="payment" should only be used for
      industry-standard payment cards. If you are
      implementing a closed-loop payment system (e.g.
      stored value cards for a specific merchant or
      transit system), use category="other". This is
      because only one "payment" card may be active at
      a time, whereas all "other" cards are active
      simultaneously (subject to AID dispatch).
  -->

  <aid-group android:description=
        "@string/card_title" android:category="other">
      <aid-filter android:name="F222222222"/>
  </aid-group>
</host-apdu-service>
```

The service class also depends on object `AccountStorage`, which for example reads as follows:

```
/**
 * Utility class for persisting account numbers to disk.
 *
 * The default SharedPreferences instance is used as
 * the backing storage. Values are cached in memory for
 * performance.
 */
object AccountStorage {
  private val PREF_ACCOUNT_NUMBER = "account_number"
  private val DEFAULT_ACCOUNT_NUMBER = "00000000"
  private val TAG = "AccountStorage"
  private var sAccount: String? = null
  private val sAccountLock = Any()

  fun setAccount(c: Context, s: String) {
      synchronized(sAccountLock) {
          Log.i(TAG, "Setting account number: $s")
          val prefs = PreferenceManager.
              getDefaultSharedPreferences(c)
          prefs.edit().
                putString(PREF_ACCOUNT_NUMBER, s).
```

```
                commit()
            sAccount = s
        }
    }

    fun getAccount(c: Context): String? {
        synchronized(sAccountLock) {
            if (sAccount == null) {
                val prefs = PreferenceManager.
                      getDefaultSharedPreferences(c)
                val account = prefs.getString(
                      PREF_ACCOUNT_NUMBER,
                      DEFAULT_ACCOUNT_NUMBER)
                sAccount = account
            }
            return sAccount
        }
    }
}
```

Android and Bluetooth

Android allows you to add your own Bluetooth functionality. An exhaustive description of all that can be done to serve Bluetooth's needs is beyond the scope of this book, but to learn how to do the following, please see the online documentation for Bluetooth in Android:

- Scan for available local Bluetooth devices (in case you have more than one)
- Scan for paired remote Bluetooth devices
- Scan for services a remote device provides
- Establish communication channels
- Transfer data between local and remote devices
- Work with profiles
- Add Bluetooth servers on your Android device

What we will do here is describe the implementation of an RfComm channel to transfer serial data between your smartphone and an external Bluetooth service. With this use case, you already have a powerful means for Bluetooth communication at hand. You can, for example, use it to control robots or smart home gadgets.

A Bluetooth RfComm Server

It is surprisingly hard on the Web to find valuable information about setting up Bluetooth servers. However, for development, it is necessary to implement a Bluetooth server so you can test the Android app. And such a test server might also serve as the basis for real-world scenarios you might think of.

A good candidate for a Bluetooth server technology is BlueCove, which is an open source project. Parts of it are licensed under an Apache License V2.0 and other parts under GPL, so while it is easy to incorporate in your own projects, you need to check whether for commercial projects the license is applicable for your needs. In the following paragraphs, I will describe how to set up a RfComm Bluetooth server on Linux using BlueCove and Groovy. For Windows, you'll have to adapt the startup script and use DLL libraries instead.

Start with downloading and installing Groovy. Any modern version should do. Next, download BlueCove. The version I tested is 2.1.0, but you might try newer versions as well. You need the files `bluecove-2.1.0.jar`, `bluecove-emu-2.1.0.jar`, and `bluecove-gpl-2.1.0.jar`. Temporarily extract the JARs as zip files somewhere and create a folder structure as follows:

```
libbluecove.jnilib
startRfComm.sh
libbluecove.so
libbluecove_x64.so
libs/
    bluecove-2.1.0.jar
    bluecove-emu-2.1.0.jar
    bluecove-gpl-2.1.0.jar
scripts/
    rfcomm.groovy
```

> **Note** Depending on the Linux distribution you use, you might need to add a symlink as follows:
>
> `cd /usr/lib/x86_64-linux-gnu/`
>
> `ln -s libbluetooth.so.3 libbluetooth.so`
>
> You must do this as root.

> **Note** In addition, still as root, execute the following:
>
> `mkdir /var/run/sdb`
>
> `chmod 777 /var/run/sdp`

> **Note** Also, to remedy compatibility issues, you must adapt the Bluetooth server process like so:
>
> `cd/etc/systemd/system/bluetooth.target.wants/`
>
> Change inside `bluetooth.service` like so:
>
> `ExecStart=/usr/lib/bluetooth/bluetoothd →`
>
> `ExecStart=/usr/lib/bluetooth/bluetoothd -C`
>
> Then enter the following in the terminal
>
> `systemctl daemon-reload` **and** `systemctl restart bluetooth`

The file `startRfComm.sh` is the startup script. Create it and inside write the following, fixing paths accordingly:

```
#!/bin/bash

export JAVA_HOME=/opt/jdk8
export GROOVY_HOME=/opt/groovy

$GROOVY_HOME/bin/groovy \
 -cp libs/bluecove-2.1.0.jar:libs/bluecove-emu-2.1.0.jar
:libs/bluecove-gpl-2.1.0.jar \
 -Dbluecove.debug=true \
 -Djava.library.path=. \
scripts/rfcomm.groovy
```

The server code lives inside `scripts/rfcomm.groovy`. Create it and insert the following content:

```
import javax.bluetooth.*
import javax.obex.*
import javax.microedition.io.*
import groovy.transform.Canonical

// Run server as root!

// setup the server to listen for connection
// retrieve the local Bluetooth device object
LocalDevice local = LocalDevice.getLocalDevice()
local.setDiscoverable(DiscoveryAgent.GIAC)

UUID uuid = new UUID(80087355)
String url = "btspp://localhost:" + uuid.toString() +
      ";name=RemoteBluetooth"
println("URI: " + url)
StreamConnectionNotifier notifier = Connector.open(url)
// waiting for connection
while(true) {
  println("waiting for connection...")
  StreamConnection connection = notifier.acceptAndOpen()
  InputStream inputStream = connection.openInputStream()
  println("waiting for input")
  while (true) {
    int command = inputStream.read()
    if(command == -1) break
    println("Command: " + command)
  }
}
```

The server must be started as root. Once you invoke `sudo ./startRfComm.sh` on a system with a Bluetooth adapter installed, the output with timestamps removed should look like this:

```
Java 1.4+ detected: 1.8.0_60; Java HotSpot(TM) 64-Bit
   Server VM; Oracle Corporation
...
localDeviceID 0
...
BlueCove version 2.1.0 on bluez
URI: btspp://localhost:04c6093b00001000800000805f9b34fb;
   name=RemoteBluetooth
open using BlueCove javax.microedition.io.Connector
...
connecting btspp://localhost:04
   c6093b00001000800000805f9b34fb;name=RemoteBluetooth
...
created SDPSession 139982379587968
...
BlueZ major verion 4 detected
...
function sdp_extract_pdu of bluez major version 4 is called
...
waiting for connection...
```

An Android RfComm Client

With the RfComm Bluetooth server process from the preceding section running, we will now develop the client for the Android platform. It is supposed to do the following:

- Provide an activity to select the remote Bluetooth device to connect to
- Provide another activity to initiate a connection and send a message to the Bluetooth RfConn server

Start with a new project and don't forget to add Kotlin support. Change the file `AndroidManifest.xml` to read as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
      "http://schemas.android.com/apk/res/android"
    package="de.pspaeth.bluetooth">

  <uses-permission android:name=
        "android.permission.BLUETOOTH_ADMIN"/>
  <uses-permission android:name=
        "android.permission.BLUETOOTH"/>
  <application
      android:allowBackup="true"
      android:icon="@mipmap/ic_launcher"
      android:label="@string/app_name"
      android:roundIcon="@mipmap/ic_launcher_round"
      android:supportsRtl="true"
```

```
      android:theme="@style/AppTheme">
      <activity android:name=".MainActivity">
          <intent-filter>
              <action android:name=
                    "android.intent.action.MAIN"/>
              <category android:name=
                    "android.intent.category.LAUNCHER"/>
          </intent-filter>
      </activity>
      <activity
          android:name=".DeviceListActivity"
          android:configChanges=
                "orientation|keyboardHidden"
          android:label="Select Device"
          android:theme=
                "@android:style/Theme.Holo.Dialog"/>
  </application>
</manifest>
```

Next create three layout files inside `res/layout`. The first, `activity_main.xml`, contains a status line and two buttons.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  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"
  android:orientation="vertical">

  <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="horizontal">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="State: " />
    <TextView
        android:id="@+id/state"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
  </LinearLayout>

  <Button
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="Scan Devices"
      android:onClick="scanDevices"/>
```

```
  <Button
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="RfComm"
      android:onClick="rfComm"/>
</LinearLayout>
```

> **Note** For simplicity I added text as literals. In a production environment, you should of course use string resources.

The next layout file, `device_list.xml`, is for the remote device selector activity:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
      "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

  <TextView
      android:id="@+id/title_paired_devices"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:background="#666"
      android:paddingLeft="5dp"
      android:text="Paired Devices"
      android:textColor="#fff"
      android:visibility="gone"
      />

  <ListView
      android:id="@+id/paired_devices"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:stackFromBottom="true"
      />

  <TextView
      android:id="@+id/title_new_devices"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:background="#666"
      android:paddingLeft="5dp"
      android:text="Other Devices"
      android:textColor="#fff"
      android:visibility="gone"
      />
```

```
  <ListView
      android:id="@+id/new_devices"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_weight="2"
      android:stackFromBottom="true"
      />

  <Button
      android:id="@+id/button_scan"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="Scan"
      />
</LinearLayout>
```

The last, `device_name.xml`, is for laying out list items from the device lister activity:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android=
        "http://schemas.android.com/apk/res/android"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:padding="5dp"
      android:textSize="18sp" />
```

The `DeviceListActvity` class is an adapted version of the device lister activity of the Bluetooth chat example from the Android developer documentation.

```
/**
 * This Activity appears as a dialog. It lists any
 * paired devices and devices detected in the area after
 * discovery. When a device is chosen by the user, the
 * MAC address of the device is sent back to the parent
 * Activity in the result Intent.
 */
class DeviceListActivity : Activity() {
  companion object {
      private val TAG = "DeviceListActivity"
      var EXTRA_DEVICE_ADDRESS = "device_address"
  }

  private var mBtAdapter: BluetoothAdapter? = null
  private var mNewDevicesArrayAdapter:
        ArrayAdapter<String>? = null
```

`OnItemClickListener` is an example for the implementation of a *single method interface* in Kotlin.

```
private val mDeviceClickListener =
      AdapterView.OnItemClickListener {
            av, v, arg2, arg3 ->
     // Cancel discovery because it's costly and we're
```

```
    // about to connect
    mBtAdapter!!.cancelDiscovery()

    // Get the device MAC address, which is the last
    // 17 chars in the View
    val info = (v as TextView).text.toString()
    val address = info.substring(info.length - 17)

    // Create the result Intent and include the MAC
    // address
    val intent = Intent()
    intent.putExtra(EXTRA_DEVICE_ADDRESS, address)

    // Set result and finish this Activity
    setResult(Activity.RESULT_OK, intent)
    finish()
}
```

The `BroadcastReceiver` listens for discovered devices and changes the title when discovery is finished.

```
/**
 * Listening for discovered devices.
 */
private val mReceiver = object : BroadcastReceiver() {
    override
    fun onReceive(context: Context, intent: Intent) {
        val action = intent.action

        // When discovery finds a device
        if (BluetoothDevice.ACTION_FOUND == action) {
            // Get the BluetoothDevice object from
            // the Intent
            val device = intent.
                  getParcelableExtra<BluetoothDevice>(
                  BluetoothDevice.EXTRA_DEVICE)
            // If it's already paired, skip it,
            // because it's been listed already
            if (device.bondState !=
                  BluetoothDevice.BOND_BONDED) {
                mNewDevicesArrayAdapter!!.add(
                      device.name + "\n" +
                      device.address)
            }
            // When discovery is finished, change the
            // Activity title
        } else if (BluetoothAdapter.
              ACTION_DISCOVERY_FINISHED == action) {
            setProgressBarIndeterminateVisibility(
                  false)
            setTitle("Select Device")
            if (mNewDevicesArrayAdapter!!.count
                  == 0) {
```

```
            val noDevices = "No device"
            mNewDevicesArrayAdapter!!.add(
                  noDevices)
        }
     }
  }
}
```

As usual, the onCreate() callback method sets up the user interface.

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

    // Setup the window
    requestWindowFeature(Window.
          FEATURE_INDETERMINATE_PROGRESS)
    setContentView(R.layout.activity_device_list)

    // Set result CANCELED in case the user backs out
    setResult(Activity.RESULT_CANCELED)

    // Initialize the button to perform device
    // discovery
    button_scan.setOnClickListener { v ->
        doDiscovery()
        v.visibility = View.GONE
    }

    // Initialize array adapters. One for already
    // paired devices and one for newly discovered
    // devices
    val pairedDevicesArrayAdapter =
          ArrayAdapter<String>(this,
                R.layout.device_name)
    mNewDevicesArrayAdapter =
          ArrayAdapter(this,
                R.layout.device_name)

    // Find and set up the ListView for paired devices
    val pairedListView = paired_devices as ListView
    pairedListView.adapter = pairedDevicesArrayAdapter
    pairedListView.onItemClickListener =
          mDeviceClickListener

    // Find and set up the ListView for newly
    // discovered devices
    val newDevicesListView = new_devices as ListView
    newDevicesListView.adapter =
          mNewDevicesArrayAdapter
    newDevicesListView.onItemClickListener =
          mDeviceClickListener
```

```
    // Register for broadcasts when a device is
    // discovered
    var filter =
          IntentFilter(BluetoothDevice.ACTION_FOUND)
    this.registerReceiver(mReceiver, filter)

    // Register for broadcasts when discovery has
    // finished
    filter = IntentFilter(BluetoothAdapter.
          ACTION_DISCOVERY_FINISHED)
    this.registerReceiver(mReceiver, filter)

    // Get the local Bluetooth adapter
    mBtAdapter = BluetoothAdapter.getDefaultAdapter()

    // Get a set of currently paired devices
    val pairedDevices = mBtAdapter!!.bondedDevices

    // If there are paired devices, add each one to
    // the ArrayAdapter
    if (pairedDevices.size > 0) {
        title_paired_devices.visibility = View.VISIBLE
        for (device in pairedDevices) {
            pairedDevicesArrayAdapter.add(
                  device.name + "\n" + device.address)
        }
    } else {
        val noDevices = "No devices"
        pairedDevicesArrayAdapter.add(noDevices)
    }
}
```

The `onDestroy()` callback method is used to clean up stuff. Finally, the `doDiscovery()` method performs the actual discovery work.

```
override fun onDestroy() {
    super.onDestroy()

    // Make sure we're not doing discovery anymore
    if (mBtAdapter != null) {
        mBtAdapter!!.cancelDiscovery()
    }

    // Unregister broadcast listeners
    this.unregisterReceiver(mReceiver)
}

/**
 * Start device discover with the BluetoothAdapter
 */
private fun doDiscovery() {
    Log.d(TAG, "doDiscovery()")
```

```
    // Indicate scanning in the title
    setProgressBarIndeterminateVisibility(true)
    setTitle("Scanning")

    // Turn on sub-title for new devices
    title_new_devices.visibility = View.VISIBLE

    // If we're already discovering, stop it
    if (mBtAdapter!!.isDiscovering) {
        mBtAdapter!!.cancelDiscovery()
    }

    // Request discover from BluetoothAdapter
    mBtAdapter!!.startDiscovery()
  }
}
```

The `MainActivity` class is responsible for checking and acquiring permissions and constructs a `BluetoothCommandService` that we will describe later.

```
class MainActivity : AppCompatActivity() {
  companion object {
      val REQUEST_ENABLE_BT = 42
      val REQUEST_QUERY_DEVICES = 142
}
var mBluetoothAdapter: BluetoothAdapter? = null
var mCommandService:BluetoothCommandService? = null
```

The `onCreate()` callback inside the activity gets used to set up the user interface and register the Bluetooth adapter.

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

    val permission1 = ContextCompat.
          checkSelfPermission(
          this, Manifest.permission.BLUETOOTH)
    val permission2 = ContextCompat.
          checkSelfPermission(
          this, Manifest.permission.BLUETOOTH_ADMIN)
    if (permission1 !=
          PackageManager.PERMISSION_GRANTED ||
        permission2 !=
          PackageManager.PERMISSION_GRANTED)
    {
        ActivityCompat.requestPermissions(this,
            arrayOf(
              Manifest.permission.BLUETOOTH,
              Manifest.permission.BLUETOOTH_ADMIN),
            642)
    }
```

```
    mBluetoothAdapter =
          BluetoothAdapter.getDefaultAdapter()

    if (mBluetoothAdapter == null) {
        Toast.makeText(this,
              "Bluetooth is not supported",
              Toast.LENGTH_LONG).show()
        finish()
    }

    if (!mBluetoothAdapter!!.isEnabled()) {
        val enableIntent = Intent(
              BluetoothAdapter.ACTION_REQUEST_ENABLE)
        startActivityForResult(
                 enableIntent, REQUEST_ENABLE_BT)
    }
}
```

The `scanDevices()` method is used for calling the system's Bluetooth device scanner.

```
/**
 * Launch the DeviceListActivity to see devices and
 * do scan
 */
fun scanDevices(v:View) {
    val serverIntent = Intent(
          this, DeviceListActivity::class.java)
    startActivityForResult(serverIntent,
          REQUEST_QUERY_DEVICES)
}
```

The methods `rfComm` and `sendMessage()` handle the sending of Bluetooth messages.

```
fun rfComm(v: View) {
    sendMessage("The message")
}

/**
 * Sends a message.
 *
 * @param message A string of text to send.
 */
private fun sendMessage(message: String) {
    if (mCommandService?.mState !==
          BluetoothCommandService.Companion.
          State.CONNECTED)
    {
        Toast.makeText(this, "Not connected",
              Toast.LENGTH_SHORT).show()
        return
    }
```

```
    // Check that there's actually something to send
    if (message.length > 0) {
        val send = message.toByteArray()
        mCommandService?.write(send)
    }
}
```

The actual connection to a device gets done from inside the method `connectDevice()`.

```
private
fun connectDevice(data: Intent, secure: Boolean) {
    val macAddress = data.extras!!
          .getString(
          DeviceListActivity.EXTRA_DEVICE_ADDRESS)
    mBluetoothAdapter?.
          getRemoteDevice(macAddress)?.run {
        val device = this
        mCommandService =
              BluetoothCommandService(
              this@MainActivity, macAddress).apply {
            addStateChangeListener { statex ->
                runOnUiThread {
                  state.text = statex.toString()
                }
            }
            connect(device)
        }
    }
}

private fun fetchUuids(device: BluetoothDevice) {
    device.fetchUuidsWithSdp()
}
```

The callback method `onActivityResult()` handles the returning from the system's device chooser. Here we just perform a connection to the device chosen.

```
override
fun onActivityResult(requestCode: Int,
      resultCode: Int, data: Intent) {
    when (requestCode) {
        REQUEST_QUERY_DEVICES -> {
            if (resultCode == Activity.RESULT_OK) {
                connectDevice(data, false)
            }
        }
      }
    }
}
```

Class `BluetoothCommandService` is, despite its name, not an Android service. It handles the communication with the Bluetooth server and reads as follows:

```
class BluetoothCommandService(context: Context,
      val macAddress:String) {
  companion object {
      // Unique UUID for this application
      private val MY_UUID_INSECURE = UUID.fromString(
            "04c6093b-0000-1000-8000-00805f9b34fb")

      // Constants that indicate the current connection
      // state
      enum class State {
          NONE,        // we're doing nothing
          LISTEN,      // listening for incoming conns
          CONNECTING,  // initiating an outgoing conn
          CONNECTED    // connected to a remote device
      }
  }

  private val mAdapter: BluetoothAdapter
  private var createSocket: CreateSocketThread? = null
  private var readWrite: SocketReadWrite? = null
  var mState: State = State.NONE

  private var stateChangeListeners =
        mutableListOf<(State)->Unit>()
  fun addStateChangeListener(l:(State)->Unit) {
      stateChangeListeners.add(l)
  }

  init {
      mAdapter = BluetoothAdapter.getDefaultAdapter()
      changeState(State.NONE)
  }
```

Its public methods are for connecting, disconnecting, and writing data.

```
/**
 * Initiate a connection to a remote device.
 *
 * @param device The BluetoothDevice to connect
 */
fun connect(device: BluetoothDevice) {
    stopThreads()

    // Start the thread to connect with the given
    // device
    createSocket = CreateSocketThread(device).apply {
        start()
    }
}
```

```
/**
 * Stop all threads
 */
fun stop() {
    stopThreads()
    changeState(State.NONE)
}

/**
 * Write to the ConnectedThread in an unsynchronized
 * manner
 *
 * @param out The bytes to write
 * @see ConnectedThread.write
 */
fun write(out: ByteArray) {
    if (mState != State.CONNECTED) return
    readWrite?.run { write(out) }
}
```

Its private methods handle the connection threads.

```
////////////////////////////////////////////////////
////////////////////////////////////////////////////

/**
 * Start the ConnectedThread to begin managing a
 * Bluetooth connection
 *
 * @param socket The BluetoothSocket on which the
 *       connection was made
 * @param device The BluetoothDevice that has been
 *       connected
 */
private fun connected(socket: BluetoothSocket,
      device: BluetoothDevice) {
    stopThreads()

    // Start the thread to perform transmissions
    readWrite = SocketReadWrite(socket).apply {
        start()
    }
}

private fun stopThreads() {
    createSocket?.run {
        cancel()
        createSocket = null
    }
    readWrite?.run {
        cancel()
        readWrite = null
   }
}
```

```
/**
 * Indicate that the connection attempt failed.
 */
private fun connectionFailed() {
    changeState(State.NONE)
}

/**
 * Indicate that the connection was lost.
 */
private fun connectionLost() {
    changeState(State.NONE)
}
```

The connection socket handling thread itself is a dedicated `Thread` implementation.

```
/**
 * This thread runs while attempting to make an
 * outgoing connection with a device. It runs straight
 * through; the connection either succeeds or fails.
 */
private inner
class CreateSocketThread(
      private val mmDevice: BluetoothDevice) :
      Thread() {
    private val mmSocket: BluetoothSocket?

    init {
        // Get a BluetoothSocket for a connection
        // with the given BluetoothDevice
        mmSocket = mmDevice.
            createInsecureRfcommSocketToServiceRecord(
            MY_UUID_INSECURE)
        changeState(Companion.State.CONNECTING)
    }

    override fun run() {
        name = "CreateSocketThread"

        // Always cancel discovery because it will
        // slow down a connection
        mAdapter.cancelDiscovery()

        // Make a connection to the BluetoothSocket
        try {
            // This is a blocking call and will only
            // return on a successful connection or an
            // exception
            mmSocket!!.connect()
        } catch (e: IOException) {
            Log.e("LOG","Connection failed", e)
            Log.e("LOG", "Maybe device does not " +
                  " expose service " + MY_UUID_INSECURE)
            // Close the socket
            mmSocket!!.close()
```

```
            connectionFailed()
                 return
            }

            // Reset the thread because we're done
            createSocket = null

            // Start the connected thread
            connected(mmSocket, mmDevice)
        }

        fun cancel() {
            mmSocket!!.close()
        }
}
```

For reading and writing data from and to the connection socket, we use another thread.

```
/**
 * This thread runs during a connection with a
 * remote device. It handles all incoming and outgoing
 * transmissions.
 */
private inner
class SocketReadWrite(val mmSocket: BluetoothSocket) :
      Thread() {
    private val mmInStream: InputStream?
    private val mmOutStream: OutputStream?

    init {
        mmInStream = mmSocket.inputStream
        mmOutStream = mmSocket.outputStream
        changeState(Companion.State.CONNECTED)
    }

    override fun run() {
        val buffer = ByteArray(1024)
        var bytex: Int

        // Keep listening to the InputStream while
        // connected
        while (mState ==
              Companion.State.CONNECTED) {

    try {
        // Read from the InputStream
        bytex = mmInStream!!.read(buffer)
    } catch (e: IOException) {
        connectionLost()
        break
    }
  }
}
```

```
    /**
     * Write to the connected OutStream.
     *
     * @param buffer The bytes to write
     */
    fun write(buffer: ByteArray) {
        mmOutStream!!.write(buffer)
    }

    fun cancel() {
        mmSocket.close()
    }
}
```

Finally, we provide a method to tell interested parties when a socket connection state changes. Here this also emits a logging statement. For production code you'd remove this or provide this information to the user some other way.

```
  private fun changeState(newState:State) {
      Log.e("LOG",
            "changing state: ${mState} -> ${newState}")
      mState = newState
      stateChangeListeners.forEach { it(newState) }
  }
}
```

> **Note** The UUID from the companion object must match the UUID you see in the server startup logs.

This class does is the following:

- Once its `connect(...)` method gets called, it starts a connection attempt.
- If the connection succeeds, another thread for initializing input and output streams using the connection object gets started. Note that the input stream is not used in this example; it is here for informational purposes.
- By virtue of its `mState` member, clients can check for the connection state.
- If connected, the method `write(...)` can be called to send data through the connection channel.

To test the connection, press the RFCOMM button on the UI. The server application should then log the following:

```
Command: 84
Command: 104
Command: 101
Command: 32
Command: 109
Command: 101
Command: 115
Command: 115
Command: 97
Command: 103
Command: 101
```

This is the numerical representation of the message “The message.”

Chapter 13

Hardware

Android can do more than present a GUI on a smartphone. Android is also about wearables, talking to appropriately equipped TV sets and infotainment in cars. Smartphones also have cameras, NFC and Bluetooth adapters, and sensors for position, movement, orientation, and fingerprints. And yes, smartphones can do phone calls as well. This chapter describes how the Android OS can run on devices other than smartphones and how to interact with the device's hardware.

Programming with Wearables

Google Wear is about small devices you wear on your body. Although right now this is pretty much restricted to smartwatches you buy and then wrap around your wrist, future devices might include your glasses, your clothes, or whatever else you might think of. Right now Google Wear using means having a smartphone with you and connecting it to a Google Wear device via some pairing mechanism, but modern devices also may run in a stand-alone fashion. This means to function they won't need paired smartphones any longer and themselves can connect to the Internet, a cellular network, or a local network via Wi-Fi, Bluetooth, or a cellular adapter.

If you happen to use a paired smartphone for a Google Wear app, this no longer is restricted to running only Android, so you can pair a Google Wear device with an Android smartphone or an Apple iOS phone. The Android Wear OS works with paired phones running Android version 4.4 or higher and iOS 9.3 or higher.

Google's design guidelines for smartphone apps (more precisely, the demand for an easy and expressive user interface) are even more important for Wear apps. Because of the limited space and input capabilities, it is absolutely vital for Wear-related development to reduce UI elements and front-end workflows to a bare minimum. Otherwise, you risk your app's usability and acceptance degrading significantly.

P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_13

The following are common use cases for Google Wear apps:

- Designing own watch faces (time and date display)
- Adding face complications (custom face elements)
- Displaying notifications
- Messaging
- Voice interaction
- Google Assistant
- Playing music
- Making and receiving calls
- Alarms
- Apps with simple user interfaces
- Companion apps to smartphone and tablet apps
- Sensor apps
- Location-based services
- Pay apps

In the following sections, we will be looking at development matters for Google Wear apps.

Wearables Development

While to develop Wear apps you can mostly use the same tools and techniques you use for smartphone or tablet app development, you have to keep in mind the limited space on smartwatches and the different way users interact with watches compared to other devices.

Nevertheless, the prominent place to start Wear development is Android Studio, and in this section we describe what to do to set up your IDE to start Wear development and how to get devices connected to Android Studio.

For developing Wear apps, we first have to point out that there are two operation modes.

- **Pairing a wearable device with a smartphone**

  Because of technical restrictions, it is not possible to pair a virtual smartwatch with a virtual smartphone. So, you have to use a real phone to pair a virtual smartwatch.

- **Stand-alone mode**

  The Wear app runs on its own, without needing to pair with a smartphone. It is highly recommended for modern apps to be able to do sensible things also in stand-alone mode.

In either case, create a new Android Studio project, and in the Target Android Devices section, select only the Wear box. As a minimum API level, choose API 24. On the subsequent screen, select one of the following:

- **Add No Activity**

  Proceed without adding an activity. You will have to do that later manually.

- **Blank Wear Activity**

  Add the following as the layout:

```
<android.support.wear.widget.BoxInsetLayout ...>
  <FrameLayout ...>
  </FrameLayout>
</android.support.wear.widget.BoxInsetLayout>
```

Add the following as an activity class:

```
class MainActivity : WearableActivity() {
  override
  fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    setAmbientEnabled() // Enables Always-on
  }
}
```

- **Google Maps Wear Activity**

  Add the following as the layout:

```
<android.support.wear.widget.
      SwipeDismissFrameLayout ...>
  <FrameLayout ...>
      <fragment android:id="@+id/map" android:name=
          "com.google.android.gms.maps.MapFragment"
      ... />
    </FrameLayout>
  </android.support.wear.widget.
        SwipeDismissFrameLayout>
```

Add the following as an activity class:

```
class MapsActivity : WearableActivity(),
      OnMapReadyCallback {
  override fun onCreate(savedState: Bundle?) {
    super.onCreate(savedState)
    setAmbientEnabled() // Enables always on
    setContentView(R.layout.activity_maps)
```

```
    // Enables the Swipe-To-Dismiss Gesture
    ...
    // Adjusts margins
    ...
  }
  override
  fun onMapReady(googleMap: GoogleMap) {
    ...
  }
}
```

- **Watch Face**

  This option does not create an activity; instead, it builds a service class needed to define a watch face.

This choice corresponds to the development paradigm you choose. You will create one of the following:

- A smartphone-like app that needs explicitly to be started on the watch to run. This includes a Google Maps app for Wear.
- A watch face. This is more or less a graphics design issue; a face is the visual appearance of time and date on the watch's surface.
- A face complication. This is a feature added to a face.

We will be talking about the different development paths in the following sections.

Next, open Tools ➤ AVD Manager and create a new virtual Wear device. You can now start your app on a virtual Wear device. Unless you chose Add No Activity, the emulator should already show a starting UI on the face.

To pair the virtual watch with your smartphone, connect the smartphone to your development PC with a USB cable, make it a development device (tap seven times on the build number at the bottom of the system settings), and then install the Wear OS using the Google app on the smartphone. On your development PC, set up the communication via this:

```
./adb -d forward tcp:5601 tcp:5601
```

Start the app, and from the menu choose Connect to Emulator.

If you want to develop using a real smartwatch and need debugging capabilities, the online resource "Debugging a Wear OS App" shows more information about how to set up a smartwatch debugging process.

Wearables App User Interface

Before you start creating a user interface for your Wear app, consider using one of the built-in mechanisms, namely, a notification or a face complication as described in the following sections. If, however, you think it is necessary your Wear app to present its own layout, do not just copy a smartphone app layout and use it for Wear. Instead, to build a

genuine Wear user interface, use the special UI elements provided by the Wear support library. To use them, make sure the module's `build.gradle` file contains the following:

```
dependencies {
  ...
  implementation 'com.android.support:wear:26.0.0'
}
```

This library contains various classes that help you to build a UI with elements especially tailored for Wear development. The page "android.support.wear.widget" in the online API documentation contains detailed information about how to use those classes.

Wearables Faces

If you want to create a Wear face showing the time and date in a certain custom design, start with the project creation wizard as described earlier, using the Watch Face option.

> **Caution** The watch face example provided in Android Studio 3.1 contains a bug. It tries to start a default activity that does not exist for a face-only app. To fix this, open Edit Configurations in the Run menu, and in Launch Options change Launch to Nothing.

The wizard service class that is generated provides a pretty elaborate example for a watch face that you can use as a starting point for your own faces.

Adding Face Complications

Face complications are placeholders for data snippets in a face. The complication data providers are strictly separated from the complication renderers, so in your face you do not say you want to show certain complications. Instead, you specify places where to show complications, and you also specify possibly complication data types, but you let the user decide which complications to show where exactly.

In this section, we talk about how to enhance your face to show complication data. For this aim, I present a minimum invasive way to update your face implementation so it will be easier for you to realize your own ideas. Having a running face as described earlier is a requirement for this section.

We start with the entries in `AndroidManifest.xml`.

- We need a way to tell Android that we are going to have a configuration activity for complication UI elements. This happens by adding this inside the `service` element (remove the line breaks indicated by ¬):

```
<meta-data
  android:name=
      "com.google.android.wearable. ¬
       watchface.wearableConfigurationAction"
  android:value=
      "com.example.xyz88.CONFIG_COMPLICATION"/>
```

This shows Android that there exists a complication administration activity. We map it to the new activity described next.

- We add the information about a permission inquiry activity and a configuration activity as follows:

```
<activity android:name=
      "android.support.wearable. ¬
      complications. ¬
      ComplicationHelperActivity"/>
<activity
    android:name=
      ".ComplicationConfigActivity"
      android:label="@string/app_name">
    <intent-filter>
      <action android:name=
          "com.example.xyz88. ¬
          CONFIG_COMPLICATION"/>
      <category android:name=
          "com.google.android. ¬
          wearable.watchface.category. ¬
          WEARABLE_CONFIGURATION"/>
      <category android:name=
          "android.intent.category. ¬
          DEFAULT"/>
    </intent-filter>
</activity>
```

Next we add the following inside the `Face` class at any suitable place:

```
lateinit var compl : MyComplications
private fun initializeComplications() {
  compl = MyComplications()
  compl.init(this@MyWatchFace, this)
}

override
fun onComplicationDataUpdate(
      complicationId: Int,
      complicationData: ComplicationData)
```

```
{
      compl.onComplicationDataUpdate(
          complicationId,complicationData)
}

private fun drawComplications(
      canvas: Canvas, drawWhen: Long) {
  compl.drawComplications(canvas, drawWhen)
}

// Fires PendingIntent associated with
// complication (if it has one).
private fun onComplicationTap(
      complicationId:Int) {
  Log.d("LOG", "onComplicationTap()")
  compl.onComplicationTap(complicationId)
}
```

In the same file, add the following to `ci.onCreate(...)`:

```
initializeComplications()
```

At the end of `onSurfaceChanged(...)`, add the following:

```
compl.updateComplicationBounds(width, height)
```

Inside the `onTapCommand(...)` function, replace the corresponding `where` block branch as follows:

```
WatchFaceService.TAP_TYPE_TAP -> {
  // The user has completed the tap gesture.
  // Toast.makeText(applicationContext, R.string.message,
  Toast.LENGTH_SHORT)
  //        .show()
  compl.getTappedComplicationId(x, y)?.run  {
    onComplicationTap(this)
  }
}
```

This figures out whether the user tapped one of the shown complications and, if this is the case, forwards the event to one of the new functions we defined. Finally, inside `onDraw(...)`, write the following:

```
...
drawBackground(canvas)
drawComplications(canvas, now)
drawWatchFace(canvas)
...
```

To handle the complications, create a new class called `MyComplications` with this content:

```
class MyComplications {
```

We first in the companion object define a couple of constants and utility methods.

```
companion object {
    fun getComplicationId(
          pos: ComplicationConfigActivity.
               ComplicationLocation): Int {
        // Add supported locations here
        return when(pos) {
            ComplicationConfigActivity.
                ComplicationLocation.LEFT ->
              LEFT_COMPLICATION_ID
            ComplicationConfigActivity.
                ComplicationLocation.RIGHT ->
              RIGHT_COMPLICATION_ID
            else -> -1
        }
    }

    fun getSupportedComplicationTypes(
          complicationLocation:
              ComplicationConfigActivity.
              ComplicationLocation): IntArray? {
        return when(complicationLocation) {
            ComplicationConfigActivity.
                  ComplicationLocation.LEFT ->
                COMPLICATION_SUPPORTED_TYPES[0]
            ComplicationConfigActivity.
                  ComplicationLocation.RIGHT ->
                COMPLICATION_SUPPORTED_TYPES[1]
            else -> IntArray(0)
        }
    }

    private val LEFT_COMPLICATION_ID = 0
    private val RIGHT_COMPLICATION_ID = 1
    val COMPLICATION_IDS = intArrayOf(
          LEFT_COMPLICATION_ID, RIGHT_COMPLICATION_ID)
    private val complicationDrawables =
          SparseArray<ComplicationDrawable>()
    private val complicationDat =
          SparseArray<ComplicationData>()

    // Left and right dial supported types.
    private val COMPLICATION_SUPPORTED_TYPES =
      arrayOf(
        intArrayOf(ComplicationData.TYPE_RANGED_VALUE,
                   ComplicationData.TYPE_ICON,
                   ComplicationData.TYPE_SHORT_TEXT,
                   ComplicationData.TYPE_SMALL_IMAGE),
```

```
        intArrayOf(ComplicationData.TYPE_RANGED_VALUE,
                   ComplicationData.TYPE_ICON,
                   ComplicationData.TYPE_SHORT_TEXT,
                   ComplicationData.TYPE_SMALL_IMAGE)
      )
}

private lateinit var ctx:CanvasWatchFaceService
private lateinit var engine:MyWatchFace.Engine
```

Inside an `init()` method we register the complications to draw. The method `onComplicationDataUpdate()` is used to handle complication data updates, and the method `updateComplicationBounds()` reacts to complication size changes.

```
fun init(ctx:CanvasWatchFaceService,
      engine: MyWatchFace.Engine) {
    this.ctx = ctx
    this.engine = engine

    // A ComplicationDrawable for each location
    val leftComplicationDrawable =
        ctx.getDrawable(custom_complication_styles)
        as ComplicationDrawable
    leftComplicationDrawable.setContext(
          ctx.applicationContext)
    val rightComplicationDrawable =
        ctx.getDrawable(custom_complication_styles)
        as ComplicationDrawable
    rightComplicationDrawable.setContext(
          ctx.applicationContext)
    complicationDrawables[LEFT_COMPLICATION_ID] =
          leftComplicationDrawable
    complicationDrawables[RIGHT_COMPLICATION_ID] =
          rightComplicationDrawable

    engine.setActiveComplications(*COMPLICATION_IDS)
}

fun onComplicationDataUpdate(
        complicationId: Int,
        complicationData: ComplicationData) {
    Log.d("LOG", "onComplicationDataUpdate() id: " +
          complicationId);
    complicationDat[complicationId] = complicationData
    complicationDrawables[complicationId].
          setComplicationData(complicationData)
    engine.invalidate()
}
```

```
fun updateComplicationBounds(width: Int,
      height: Int) {
    // For most Wear devices width and height
    // are the same
    val sizeOfComplication = width / 4
    val midpointOfScreen = width / 2

    val horizontalOffset =
          (midpointOfScreen - sizeOfComplication) / 2
    val verticalOffset =
          midpointOfScreen - sizeOfComplication / 2

    complicationDrawables.get(LEFT_COMPLICATION_ID).
          bounds =
            // Left, Top, Right, Bottom
            Rect(
              horizontalOffset,
              verticalOffset,
              horizontalOffset + sizeOfComplication,
              verticalOffset + sizeOfComplication)
    complicationDrawables.get(RIGHT_COMPLICATION_ID).
          bounds =
            // Left, Top, Right, Bottom
            Rect(
              midpointOfScreen + horizontalOffset,
              verticalOffset,
              midpointOfScreen + horizontalOffset +
                    sizeOfComplication,
              verticalOffset + sizeOfComplication)
}
```

The method `drawComplications()` actually draws the complications. For this aim, we scan through the complications we registered inside the `init` block.

```
fun drawComplications(canvas: Canvas, drawWhen: Long) {
    COMPLICATION_IDS.forEach {
        complicationDrawables[it].
              draw(canvas, drawWhen)
    }
}
```

We need the ability to find out whether one of our complications has been tapped. The method `getTappedComplicationId()` is responsible for that. Finally, a method `onComplicationTap()` reacts to such events.

```
// Determines if tap happened inside a complication
// area, or else returns null.
fun getTappedComplicationId(x:Int, y:Int):Int? {
    val currentTimeMillis = System.currentTimeMillis()
    for(complicationId in
          MyComplications.COMPLICATION_IDS) {
        val res =
              complicationDat[complicationId]?.run {
```

```
            var res2 = -1
            if(isActive(currentTimeMillis)
                && (getType() !=
                  ComplicationData.TYPE_NOT_CONFIGURED)
                && (getType() !=
                  ComplicationData.TYPE_EMPTY))
            {
                val complicationDrawable =
                  complicationDrawables[complicationId]
                val complicationBoundingRect =
                  complicationDrawable.bounds
                if (complicationBoundingRect.width()
                      > 0) {
                    if (complicationBoundingRect.
                        contains(x, y)) {
                    res2 = complicationId
                    }
              } else {
                  Log.e("LOG",
                  "Unrecognized complication id.")
              }
          }
          res2
      } ?: -1
      if(res != -1) return res
    }
    return null
}

// The user tapped on a complication
fun onComplicationTap(complicationId:Int) {
    Log.d("LOG", "onComplicationTap()")

    val complicationData =
          complicationDat[complicationId]
    if (complicationData != null) {
        if (complicationData.getTapAction()
              != null) {
            try {
                complicationData.getTapAction().send()
            } catch (e: Exception ) {
                Log.e("LOG",
                  "onComplicationTap() tap error: " +
                  e);
          }
       } else if (complicationData.getType() ==
             ComplicationData.TYPE_NO_PERMISSION) {
          // Launch permission request.
          val componentName = ComponentName(
                  ctx.applicationContext,
                  MyComplications::class.java)
```

```
          val permissionRequestIntent =
            ComplicationHelperActivity.
            createPermissionRequestHelperIntent(
                  ctx.applicationContext,
                  componentName)
            ctx.startActivity(permissionRequestIntent)
            }

        } else {
            Log.d("LOG",
                "No PendingIntent for complication " +
                complicationId + ".")
        }
    }
}
```

What is left to do is write the configuration activity. For that purpose, create a new Kotlin class called `ComplicationConfigActivity` with the following content:

```
class ComplicationConfigActivity :
      Activity(), View.OnClickListener {
  companion object {
      val TAG = "LOG"
      val COMPLICATION_CONFIG_REQUEST_CODE = 1001
  }

  var mLeftComplicationId: Int = 0
  var mRightComplicationId: Int = 0
  var mSelectedComplicationId: Int = 0

  // Used to identify a specific service that renders
  // the watch face.
  var mWatchFaceComponentName: ComponentName? = null

  // Required to retrieve complication data from watch
  // face for preview.
  var mProviderInfoRetriever:
        ProviderInfoRetriever? = null

  var mLeftComplicationBackground: ImageView? = null
  var mRightComplicationBackground: ImageView? = null

  var mLeftComplication: ImageButton? = null
  var mRightComplication: ImageButton? = null

  var mDefaultAddComplicationDrawable: Drawable? = null

enum class ComplicationLocation {
    LEFT,
    RIGHT
}
```

As usual, we use the `onCreate()` and `onDestroy()` callbacks to set up or clean up our user interface. Also, the method `retrieveInitialComplicationsData()` gets used by `onCreate()` to initialize the complications.

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

    setContentView(R.layout.activity_config)

    mDefaultAddComplicationDrawable =
          getDrawable(R.drawable.add_complication)

    mSelectedComplicationId = -1

    mLeftComplicationId =
          MyComplications.getComplicationId(
          ComplicationLocation.LEFT)
    mRightComplicationId =
          MyComplications.getComplicationId(
          ComplicationLocation.RIGHT)

    mWatchFaceComponentName =
          ComponentName(applicationContext,
          MyWatchFace::class.java!!)

    // Sets up left complication preview.
    mLeftComplicationBackground =
          left_complication_background
    mLeftComplication = left_complication
    mLeftComplication!!.setOnClickListener(this)

    // Sets default as "Add Complication" icon.
    mLeftComplication!!.setImageDrawable(
          mDefaultAddComplicationDrawable)
    mLeftComplicationBackground!!.setVisibility(
          View.INVISIBLE)

    // Sets up right complication preview.
    mRightComplicationBackground =
          right_complication_background
    mRightComplication = right_complication
    mRightComplication!!.setOnClickListener(this)

    // Sets default as "Add Complication" icon.
    mRightComplication!!.setImageDrawable(
          mDefaultAddComplicationDrawable)
    mRightComplicationBackground!!.setVisibility(
          View.INVISIBLE)
```

```
    mProviderInfoRetriever =
          ProviderInfoRetriever(applicationContext,
          Executors.newCachedThreadPool())
    mProviderInfoRetriever!!.init()

    retrieveInitialComplicationsData()
}

override fun onDestroy() {
    super.onDestroy()
    mProviderInfoRetriever!!.release()
}

fun retrieveInitialComplicationsData() {
    val complicationIds =
          MyComplications.COMPLICATION_IDS
    mProviderInfoRetriever!!.retrieveProviderInfo(
            object : ProviderInfoRetriever.
                  OnProviderInfoReceivedCallback() {
                override fun onProviderInfoReceived(
                      watchFaceComplicationId:
                        Int,
                      complicationProviderInfo:
                        ComplicationProviderInfo?)
                {
                    Log.d(TAG,
                        "onProviderInfoReceived: " +
                        complicationProviderInfo)
                    updateComplicationViews(
                          watchFaceComplicationId,
                          complicationProviderInfo)
                }
            },
            mWatchFaceComponentName,
            *complicationIds)
}
```

The methods `onClick()` and `launchComplicationHelperActivity()` are used to handle on-complication taps.

```
override
fun onClick(view: View) {
    if (view.equals(mLeftComplication)) {
        Log.d(TAG, "Left Complication click()")
        launchComplicationHelperActivity(
              ComplicationLocation.LEFT)
    } else if (view.equals(mRightComplication)) {
        Log.d(TAG, "Right Complication click()")
        launchComplicationHelperActivity(
              ComplicationLocation.RIGHT)
    }
}
```

```
fun launchComplicationHelperActivity(
      complicationLocation: ComplicationLocation) {

    mSelectedComplicationId =
          MyComplications.getComplicationId(
          complicationLocation)

    if (mSelectedComplicationId >= 0) {
        val supportedTypes = MyComplications.
              getSupportedComplicationTypes(
              complicationLocation)!!

        startActivityForResult(
                ComplicationHelperActivity.
                createProviderChooserHelperIntent(
                        applicationContext,
                        mWatchFaceComponentName,
                        mSelectedComplicationId,
                        *supportedTypes),
                ComplicationConfigActivity.
                COMPLICATION_CONFIG_REQUEST_CODE)
    } else {
        Log.d(TAG,
          "Complication not supported by watch face.")
    }
}
```

To handle updates that we get signaled by the Android OS, we provide the methods `updateComplicationViews()` and `onActivityResult()`.

```
fun updateComplicationViews(
        watchFaceComplicationId:
          Int,
        complicationProviderInfo:
          ComplicationProviderInfo?)
{
    Log.d(TAG, "updateComplicationViews(): id: "+
          watchFaceComplicationId)
    Log.d(TAG, "\tinfo: " + complicationProviderInfo)

    if (watchFaceComplicationId ==
          mLeftComplicationId) {
        if (complicationProviderInfo != null) {
            mLeftComplication!!.setImageIcon(
                complicationProviderInfo.providerIcon)
            mLeftComplicationBackground!!.
                setVisibility(View.VISIBLE)
        } else {
            mLeftComplication!!.setImageDrawable(
                mDefaultAddComplicationDrawable)
            mLeftComplicationBackground!!.
                setVisibility(View.INVISIBLE)
        }
```

```
    } else if (watchFaceComplicationId ==
          mRightComplicationId) {
        if (complicationProviderInfo != null) {
            mRightComplication!!.
                setImageIcon(
                complicationProviderInfo.providerIcon)
            mRightComplicationBackground!!.
                setVisibility(View.VISIBLE)

        } else {
            mRightComplication!!.setImageDrawable(
                mDefaultAddComplicationDrawable)
            mRightComplicationBackground!!.
                setVisibility(View.INVISIBLE)
        }
    }
}

override
fun onActivityResult(requestCode: Int,
      resultCode: Int, data: Intent) {
    if (requestCode ==
          COMPLICATION_CONFIG_REQUEST_CODE
        && resultCode == Activity.RESULT_OK) {

        // Retrieves information for selected
        //  Complication provider.
        val complicationProviderInfo =
            data.getParcelableExtra<
                ComplicationProviderInfo>(
                ProviderChooserIntent.
                EXTRA_PROVIDER_INFO)
        Log.d(TAG, "Provider: " +
            complicationProviderInfo)

        if (mSelectedComplicationId >= 0) {
            updateComplicationViews(
                mSelectedComplicationId,
                complicationProviderInfo)
        }
    }
  }
}
```

Note that we added a couple of logging statements that you might want to remove for productive code. A corresponding layout may read as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android=
      "http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
```

```
<View
    android:id="@+id/watch_face_background"
    android:layout_width="180dp"
    android:layout_height="180dp"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:background=
      "@drawable/settings_face_preview_background"/>

<View
    android:id="@+id/watch_face_highlight"
    android:layout_width="180dp"
    android:layout_height="180dp"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:background=
      "@drawable/settings_face_preview_highlight"/>

<View
    android:id="@+id/watch_face_arms_and_ticks"
    android:layout_width="180dp"
    android:layout_height="180dp"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:background=
      "@drawable/settings_face_preview_arms_n_ticks"/>

<ImageView
    android:id="@+id/left_complication_background"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/added_complication"
    style="?android:borderlessButtonStyle"
    android:background="@android:color/transparent"
    android:layout_centerVertical="true"
    android:layout_alignStart=
        "@+id/watch_face_background"/>

<ImageButton
    android:id="@+id/left_complication"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="?android:borderlessButtonStyle"
    android:background="@android:color/transparent"
    android:layout_alignTop=
        "@+id/left_complication_background"
    android:layout_alignStart=
        "@+id/watch_face_background"/>
```

```
<ImageView
    android:id="@+id/right_complication_background"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/added_complication"
    style="?android:borderlessButtonStyle"
    android:background="@android:color/transparent"
    android:layout_alignTop=
        "@+id/left_complication_background"
    android:layout_alignStart=
        "@+id/right_complication"/>

<ImageButton
    android:id="@+id/right_complication"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="?android:borderlessButtonStyle"
    android:background="@android:color/transparent"
    android:layout_alignTop=
        "@+id/right_complication_background"
    android:layout_alignEnd=
        "@+id/watch_face_background"/>
</RelativeLayout>
```

With all these additions, the face provides for two possible complications to be added on user demand. More complication positions are possible; just rewrite the appropriate parts of the code.

> **Note** Entering the code as shown here, Android Studio will complain about missing resources, especially drawables. For the code presented here to run, you must provide the missing resources. You can usually figure out what they get used for by looking at the names.

Providing Complication Data

A Google Wear device by default includes several complication data providers, so the user can choose among them to fill the complication placeholders in a face.

If you want to create your own complication data provider, prepare a new service as declared in `AndroidManifest.xml`.

```
<service
    android:name=".CustomComplicationProviderService"
    android:icon="@drawable/ic_watch_white"
    android:label="Service label"
    android:permission="com.google.android.wearable. ¬
          permission.BIND_COMPLICATION_PROVIDER">

  <intent-filter>
    <action android:name="android.support.wearable. ¬
          complications. ¬
          ACTION_COMPLICATION_UPDATE_REQUEST"/>
  </intent-filter>
```

```
  <meta-data
        android:name="android.support.wearable.¬
              complications.SUPPORTED_TYPES"
        android:value=
              "SHORT_TEXT,LONG_TEXT,RANGED_VALUE"/>

          <!--
          UPDATE_PERIOD_SECONDS specifies how
          often you want the system to check for updates
          to the data. A zero value means you will
           instead manually trigger updates.

           If not zero, set the interval in the order
           of minutes. The actual update may however
          differ - the system might have its own idea.
          -->
          <meta-data
              android:name="android.support.wearable.¬
                    complications.UPDATE_PERIOD_SECONDS"
              android:value="0"/>

      </service>
```

Start with a service class `CustomComplicationProviderService` as follows:

```
class CustomComplicationProviderService :
      ComplicationProviderService() {
  // This method is for any one-time per complication set
 -up.
  override
  fun onComplicationActivated(
          complicationId: Int, dataType: Int,
         complicationManager: ComplicationManager?) {
      Log.d(TAG,
          "onComplicationActivated(): $complicationId")
  }

  // The complication needs updated data from your
  // provider. Could happen because of one of:
  //   1. An active watch face complication is changed
  //      to use this provider
  //   2. A complication using this provider becomes
  //      active
  //   3. The UPDATE_PERIOD_SECONDS (manifest) has
  //      elapsed
  //   4. Manually: an update via
  //      ProviderUpdateRequester.requestUpdate()
  override fun onComplicationUpdate(
          complicationId: Int, dataType: Int,
          complicationManager: ComplicationManager) {
      Log.d(TAG,
```

```
        "onComplicationUpdate() $complicationId")

      // ... add code for data generation ...

      var complicationData: ComplicationData? = null
      when (dataType) {
          ComplicationData.TYPE_SHORT_TEXT ->
             complicationData = ComplicationData.
                Builder(ComplicationData.TYPE_SHORT_TEXT)
                  . ... create datum ...
                  .build()
          ComplicationData.TYPE_LONG_TEXT ->
            complicationData = ComplicationData.
                Builder(ComplicationData.TYPE_LONG_TEXT)
                  ...
          ComplicationData.TYPE_RANGED_VALUE ->
             complicationData = ComplicationData.
                Builder(ComplicationData.
                        TYPE_RANGED_VALUE)
                  ...
          else ->
            Log.w("LOG",
            "Unexpected complication type $dataType")
}

      if (complicationData != null) {
           complicationManager.updateComplicationData(
                complicationId, complicationData)
      } else {
          // Even if no data is sent, we inform the
          // ComplicationManager
          complicationManager.noUpdateRequired(
                 complicationId)
      }
  }

  override
  fun onComplicationDeactivated(complicationId: Int) {
      Log.d("LOG",
        "onComplicationDeactivated(): $complicationId")
  }
}
```

To manually fire requests for the system to inquire about new complication data, you use the `ProviderUpdateRequester` class as follows:

```
val compName =
  ComponentName(applicationContext,
      MyService::class.java)

val providerUpdateRequester =
  ProviderUpdateRequester(
      applicationContext, componentName)
```

```
providerUpdateRequester.requestUpdate(
    complicationId)
// To instead all complications, instead use
// providerUpdateRequester.requestUpdateAll()
```

Notifications on Wearables

Notifications on wearables can run in bridged mode and in stand-alone mode. In bridged mode, notifications get automatically synchronized with a paired smartphone; in stand-alone mode, the Wear device shows notifications independently.

To start creating your own notifications, begin with a Wear project using a blank Wear activity in the project setup wizard. Then, inside the module's `build.gradle` file, update the dependencies to read as follows (with the line breaks at ¬ removed):

```
dependencies {
  implementation fileTree(dir: 'libs', include: ['*.jar
 '])
  implementation "org.jetbrains.kotlin: ¬
      kotlin-stdlib-jre7:$kotlin_version"
  implementation ¬
      'com.google.android.support:wearable:2.3.0'
  implementation 'com.google.android.gms: ¬
      play-services-wearable:12.0.1'
  implementation ¬
      'com.android.support:percent:27.1.1'
  implementation ¬
      'com.android.support:support-v13:27.1.1'
  implementation ¬
      'com.android.support:recyclerview-v7:27.1.1'
  implementation ¬
      'com.android.support:wear:27.1.1'
compileOnly ¬
      'com.google.android.wearable:wearable:2.3.0'
}
```

Change the layout file to add a button for creating a notification, as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.wear.widget.BoxInsetLayout
  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"
  android:background="@color/dark_grey"
  android:padding="@dimen/box_inset_layout_padding"
  tools:context=".MainActivity"
  tools:deviceIds="wear">
```

```
  <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:padding=
          "@dimen/inner_frame_layout_padding"
      app:boxedEdges="all"
      android:orientation="vertical">

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

      <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Go" android:onClick="go"/>

  </LinearLayout>
</android.support.wear.widget.BoxInsetLayout>
```

The activity gets a function to react to the button press. Inside, we create and send a notification.

```
class MainActivity : WearableActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(activity_main)
      setAmbientEnabled() // Enables Always-on
  }

  fun go(v: View) {
      val notificationId = 1

      // The channel ID of the notification.
      val id = "my_channel_01"
      if (Build.VERSION.SDK_INT >=
            Build.VERSION_CODES.O) {
          // Create the NotificationChannel
          val name = "My channel"
          val description = "Channel description"
          val importance =
                NotificationManager.IMPORTANCE_DEFAULT
          val mChannel = NotificationChannel(
                 id, name, importance)
          mChannel.description = description
          // Register the channel with the system
          val notificationManager = getSystemService(
                 Context.NOTIFICATION_SERVICE)
              as NotificationManager
```

```
            notificationManager.
                  createNotificationChannel(mChannel)
        }

        // Notification channel ID is ignored for Android
        // 7.1.1 (API level 25) and lower.
        val notificationBuilder =
              NotificationCompat.Builder(this, id)
            .setSmallIcon(android.R.drawable.ic_media_play)
            .setContentTitle("Title")
            .setContentText("Content Text")

        // Get NotificationManager service
        val notificationManager =
              NotificationManagerCompat.from(this)

        // Issue the notification
        notificationManager.notify(
              notificationId, notificationBuilder.build())
    }
}
```

If you start this app, it shows a simple UI with a text and a button. Pressing the button leads to shortly displaying the notification icon, which is a "play" rectangle in our example. Using the back button and swiping up, the notification shows up with the title and contents. Also, the face your user uses might have a notification preview added. See Figure 13-1.

***Figure 13-1.** Notification on Wear*

You can also add a `PendingIntent` to your code and register it with `setContentIntent(...)` inside the builder to allow for sending an intent once the user clicks an appearing notification. Also, inside the builder, you can add action icons by using `addAction(...)` or `addActions(...)`.

Wearable-specific features can be added to a notification by constructing a `NotificationCompat.WearableExtender` object and calling `extent(...)` on the builder passing this extender object. Note that by adding actions to the `WearbleExtender` object instead of the builder, you can make sure the actions show up only on a wearable.

To add voice features to Wear notifications using predefined text responses and special features to be used in the bridge mode, please see the online documentation for wearable notifications.

Controlling App Visibility on Wearables

Wear OS devices since Android 5.1 allow for running Wear apps in the foreground even when the power-saving, or *ambient*, mode is engaged. You have two options for handling the ambient mode.

- Use the `AmbientModeSupport` class.
- Use the `WearableActivity` class.

To use the `AmbientModeSupport` class, implement a subclass of `Activity`, implement the `AmbientCallbackProvider` interface, and declare and save `AmbientController` as follows:

```
class MainActivity : FragmentActivity(),
      AmbientModeSupport.AmbientCallbackProvider {
  override
  fun getAmbientCallback():
        AmbientModeSupport.AmbientCallback

  {
     ...
  }

  lateinit
  var mAmbientController:
        AmbientModeSupport.AmbientController

  override
  fun onCreate(savedInstanceState:Bundle?) {
      super.onCreate(savedInstanceState)
      ...
      mAmbientController =
          AmbientModeSupport.attach(this)
  }
}
```

Inside the `getAmbientCallback()` function, create and return a subclass of `AmbientModeSupport.AmbientCallback`. This callback is then responsible for the switch between the standard and ambient modes. What ambient mode actually does is up to you as a developer, but you should engage power-saving measures such as dimmed and black-and-white graphics, augmented update intervals, and so on.

The second possibility to allow for ambient mode is to let your activity inherit from class `WearableActivity`, call `setAmbientEnabled()` in its `onCreate(...)` callback, and overwrite `onEnterAmbient()` and `onExitAmbient()`. If you also overwrite `onUpdateAmbient()`, you can put your screen-updating logic there and let the system decide which update frequency to use in ambient mode.

Authentication in Wear

With Wear apps being able to run in stand-alone mode, authentication becomes more important for Wear apps. Describing the appropriate procedures for this matter are out of scope for this book, but the page "Authentication in Wear" in the online documentation gives you detailed information about authentication in Wear.

Voice Capabilities in Wear

Adding voice capabilities to Wear devices makes a lot of sense since other methods for user input are limited because of the small device dimensions. You have two options: connect your app to one or more of the system-provided voice actions or define your own actions.

> **Caution** The Wear emulator cannot handle voice commands; you have to use real devices to test it.

Connecting system voice events to activities that your app provides is easy. All you have to do is to add an intent filter to your activity as follows:

```
<intent-filter>
    <action android:name=
          "android.intent.action.SEND" />
    <category android:name=
          "com.google.android.voicesearch.SELF_NOTE" />
</intent-filter>
```

Table 13-1 lists the possible voice keys.

Table 13-1. System Voice Commands

Command	Manifest	Key Extras
"OK Google, get me a taxi" "OK Google, call me a car"	`com.google.android.gms.actions.RESERVE_TAXI_RESERVATION`	

"OK Google, take a note"
"OK Google, note to self"
	`android.intent.action.SEND`
	Category:
	`com.android.voicesearch.SELF_NOTE`
		`android.content.Intent.EXTRA_TEXT`: A string with note body

"OK Google, set an alarm for 8 a.m." "OK Google, wake me up at 6 tomorrow"	`android.intent.action.SET_ALARM`	`android.provider.AlarmClock.EXTRA_HOUR`: An integer with the hour of the alarm `android.provider.AlarmClock.EXTRA_MINUTES`: An integer with the minute of the alarm

(*continued*)

Table 13-1. (*continued*)

Command	Manifest	Key Extras
"OK Google, set a timer for 10 minutes"	`android.intent.action.SET_TIMER`	`android.provider.AlarmClock.EXTRA_LENGTH`: An integer in the range of 1 to 86400 (number of seconds in 24 hours) representing the length of the timer
"OK Google, start stopwatch"	`com.google.android.wearable.action.STOPWATCH`	

"OK Google, start cycling"
"OK Google, start my bike ride"
"OK Google, stop cycling"
	`vnd.google.fitness.TRACK`
	MIME type:
	`vnd.google.fitness.activity/biking`
		`actionStatus`: A string with a value of `ActiveActionStatus` when starting and `CompletedActionStatus` when stopping

"OK Google, track my run"
"OK Google, start running"
"OK Google, stop running"
	`vnd.google.fitness.TRACK`
	MIME type:
	`vnd.google.fitness.activity/running`
		`actionStatus`: A string with a value of `ActiveActionStatus` when starting and `CompletedActionStatus` when stopping

"OK Google, start a workout"
"OK Google, track my workout"
"OK Google, stop workout"
	`vnd.google.fitness.TRACK`
	MIME type:
	`vnd.google.fitness.activity/other`
		`actionStatus`: A string with the value `ActiveActionStatus` when starting and `CompletedActionStatus` when stopping

"OK Google, what's my heart rate?"
"OK Google, what's my bpm?"
	`vnd.google.fitness.VIEW`
	MIME type:
	`vnd.google.fitness.data_type/com.google.heart_rate.bpm`	

"OK Google, how many steps have I taken?"
"OK Google, what's my step count?"
	`vnd.google.fitness.VIEW`
	MIME type:
	`vnd.google.fitness.data_type/com.google.step_count.cumulative`	

Extra data can be extracted from incoming intents as usual via one of the various `Intent.get*Extra(...)` methods.

You may also provide app-defined voice actions that can start custom activities. To do so, in `AndroidManifest.xml` define each `<action>` element in question as follows:

```
<activity android:name="MyActivity" android:label="
 MyRunningApp">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"
  />
```

```
        <category android:name="android.intent.category.
  LAUNCHER" />
     </intent-filter>
</activity>
```

This by virtue of the `label` attribute will allow you to say "Start MyRunningApp" to start the activity.

You can also let speech recognition fill in the edit fields. For this purpose, write the following:

```
val SPEECH_REQUEST_CODE = 42
val intent = Intent(
      RecognizerIntent.ACTION_RECOGNIZE_SPEECH).apply {
  putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
        RecognizerIntent.LANGUAGE_MODEL_FREE_FORM)
}.run {
  startActivityForResult(this, SPEECH_REQUEST_CODE)
}
```

Then fetch the result in the overwritten `onActivityResult(...)` callback.

```
fun onActivityResult(requestCode:Int, resultCode:Int,
      data:Intent) {
  if (requestCode and 0xFFFF == SPEECH_REQUEST_CODE
        && resultCode == RESULT_OK) {
      val results = data.getStringArrayListExtra(
               RecognizerIntent.EXTRA_RESULTS)
      String spokenText = results[0]
      // ... do something with spoken text
  }
  super.onActivityResult(
        requestCode, resultCode, data)
}
```

Speakers on Wearables

If you want to use the speakers connected to a Wear device to play some audio, you first check whether the Wear app can connect to speakers.

```
fun hasSpeakers(): Boolean {
    val packageManager = context.getPackageManager()
    val audioManager =
          context.getSystemService(
          Context.AUDIO_SERVICE) as AudioManager

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        // Check FEATURE_AUDIO_OUTPUT to guard against
        // false positives.
        if (!packageManager.hasSystemFeature(
              PackageManager.FEATURE_AUDIO_OUTPUT)) {
            return false
        }
```

```
        val devices =
               audioManager.getDevices(
               AudioManager.GET_DEVICES_OUTPUTS)
        for (device in devices) {
            if (device.type ==
                  AudioDeviceInfo.TYPE_BUILTIN_SPEAKER) {
                return true
            }
        }
    }
    return false
}
```

You can then play sound the same way as for any other app on any other device. This is described in detail in the section “Playing Audio.”

Location in Wear

To use location detection in a Wear device, you must first check whether the location data is available.

```
fun hasGps():Boolean {
  return packageManager.hasSystemFeature(
         PackageManager.FEATURE_LOCATION_GPS);
}
```

If the wearable has no own location sensor, you must instead constantly check whether the wearable is connected. You do so by handling callbacks as follows:

```
var wearableConnected = false
fun onCreate(savedInstanceState: Bundle?) {
    ...
    Wearable.getNodeClient(this@MainActivity).
          connectedNodes.addOnSuccessListener {
        wearableConnected = it.any {
             it.isNearby
        }
    }.addOnCompleteListener {
    }.addOnFailureListener {
       ...
    }
}
```

Starting from there you can handle location detection using the fused location provider as described in Chapter 8.

Data Communication in Wear

Data communication in Wear OS happens in one of two ways.

- *Direct network communication*: This is for Wear devices able to connect to a network by themselves, wanting to talk to nonpaired devices.
- *Using the wearable data layer API*: This is for communication to paired handheld devices.

For a direct network communication, use class `android.net.ConnectivityManager` for both checking for capabilities such as bandwidth and requesting new capabilities such as increased bandwidth. Please see the classes in the online API documentation for details. To actually perform network communication, use the classes and interfaces from package `android.net`.

The rest of this section is for describing the *wearable data layer API* for communication to paired handhelds.

To access the wearable data layer API, retrieve a `DataClient` or `MessageClient` via the following from inside an activity:

```
val dataClient = Wearable.getDataClient(this)
val msgClient = Wearable.getMessageClient(this)
```

You can do this often because both calls are inexpensive. A message client is best used for data with a small payload; for a larger payload, use a data client instead. Also, a data client is a reliable means of synchronizing data between Wear devices and handhelds, while the message client use a fire-and-forget mode. A message client thus does not know whether sent data actually arrives.

For sending data items using a data client, create a `PutDataMapRequest` object, call `getDataMap()` on it, and use one of the various `put...()` methods to add data. Finally, call `asPutDataRequest()` and use its result to call `DataClient.putDataItem(...)`. The latter starts the synchronization with other devices and returns a `com.google.android.gms.tasks.Task` object to which you can add listeners to watch the communication.

On the receiver side, you can observe data synchronization by extending your activity with `DataClient.OnDataChangedListener` and implementing the `fun onDataChanged(dataEvents:DataEventBuffer)` function.

For larger binary data sets like images, you can use an `Asset` as a data type to be sent over the data client, as follows:

```
fun createAssetFromBitmap(bitmap: Bitmap): Asset {
  val byteStream = ByteArrayOutputStream()
  bitmap.compress(Bitmap.CompressFormat.PNG, 100,
        byteStream)
  return Asset.createFromBytes(byteStream.
         toByteArray())
}
val bitmap = BitmapFactory.decodeResource(
      getResources(), android.R.drawable.ic_media_play)
val asset = createAssetFromBitmap(bitmap)
```

```
val dataMap = PutDataMapRequest.create("/image")
dataMap.getDataMap().putAsset("profileImage", asset)
val request = dataMap.asPutDataRequest()
val putTask: Task<DataItem> =
      Wearable.getDataClient(this).putDataItem(request)
```

To instead use a message client, we first need to find suitable message receivers. You do so by first assigning capabilities to suitable handheld apps. This can be accomplished by adding a file `wear.xml` to `res/values` with the following content:

```
<resources>
  <string-array name="android_wear_capabilities">
      <item>my_capability1</item>
      <item>my_capability2</item>
      ...
  </string-array>
</resources>
```

To find a handheld (or network node) with suitable capabilities and then send messages to it, you write the following:

```
val capabilityInfo = Tasks.await(
    Wearable.getCapabilityClient(this).getCapability(
       "my_capability1",
       CapabilityClient.FILTER_REACHABLE))
capabilityInfo.nodes.find {
  it.isNearby
}?.run {
  msgClient.sendMessage(
        this.id,"/msg/path","Hello".toByteArray())
}
```

Instead of this, you could also add a `CapabilityClient.OnCapabilityChangedListener` listener directly to the client as follows:

```
Wearable.getCapabilityClient(this).addListener({
  it.nodes.find {
    it.isNearby
  }?.run {
    msgClient.sendMessage(
       this.id,"/msg/path","Hello".toByteArray())
  }
}, "my_capability1")
```

To receive such a message, anywhere in an app installed on a handheld, register a message event listener via the following:

```
Wearable.getMessageClient(this).addListener {
      messageEvent ->
    // do s.th. with the message event
}
```

Programming with Android TV

App development targeting an Android TV device does not substantially differ from development on smartphones. However, because of the user expectation coming from decades of TV consumption, conventions are stricter compared to smartphones. Fortunately, the project builder wizard of Android Studio helps you get started. And in this section too we will be talking about important aspects of Android TV development.

Android TV Use Cases

The following are typical use cases for an Android TV app:

- Playback of video and music data streams and files
- A catalog to help users find content
- A game that can be played on Android TV (without touchscreen)
- Presenting channels with content

Starting an Android TV Studio Project

If you start a new Android TV project in Android Studio, the following are the prominent points of interest:

- Inside the manifest file, the items will make sure the app *could* also work on a smartphone with a touchscreen and that the leanback user interface needed by Android TV gets included.

```
<uses-feature
    android:name="android.hardware.touchscreen"
    android:required="false"/>
<uses-feature
    android:name="android.software.leanback"
    android:required="true"/>
```

- Still inside the manifest file, you will see the start activity to have an intent filter like this:

```
<intent-filter>
  <action android:name=
     "android.intent.action.MAIN"/>
  <category android:name=
     "android.intent.category.LEANBACK_LAUNCHER"/>
</intent-filter>
```

The category shown here is important; otherwise, Android TV will not properly recognize the app. The activity also needs to have an `android:banner` attribute, which points to a banner prominently shown on the Android TV user interface.

- Inside the module's `build.gradle` file, the leanback support library gets added inside the `dependencies` section.

```
 implementation 'com.android.support:leanback-
v17:27.1.1'
```

For development you can either use a virtual or use a real device. Virtual devices get installed via the AVD Manager in the Tools menu. For real devices, tap seven times on the build number in Settings ➤ Device ➤ About. Then in Settings, go to Preferences and enable debugging in the Developer Options.

Android TV Hardware Features

To find out whether an app is running on an Android TV, you can use `UiModeManager` as follows:

```
val isRunnigOnTv =
    (getSystemService(Context.UI_MODE_SERVICE)
         as UiModeManager).currentModeType ==
    Configuration.UI_MODE_TYPE_TELEVISION
```

Also, available features vary from device to device. If your app needs certain hardware features, you can check the availability as follows:

```
getPackageManager().
      hasSystemFeature(PackageManager.FEATURE_*)
```

For all possible features, see the API documentation of `PackageManager`.

User input on Android TV devices normally happens via a D-pad controller. To build stable apps, you should react on changes of the availability of the D-pad controller. So, inside the `AndroidManifest.xml` file, add `android: configChanges = "keyboard|keyboardHidden|navigation"` as an activity attribute. The app then gets informed about configuration changes via the overwritten callback function `fun onConfigurationChanged( newConfig : Configuration )`.

UI Development for Android TV

For Android TV development, using the *leanback theme* is suggested. For this aim, replace the `theme` attribute in the `<application>` element of the `AndroidManifest.xml` file with this:

```
android:theme="@style/Theme.Leanback"
```

This implies not using an action bar, which makes sense since Android TV does not support action bars. Also, the activity must not extend `AppCompatActivity`; instead, extend `android.support.v4.app.FragmentActivity`.

Another specialty of Android TV apps is that an occasional overscan might happen. Depending on the pixel size and aspect ratio, Android TV might clip away parts of the screen. To avoid your layout from being destroyed, adding a margin of 48dp × 27dp to the main container is suggested, as follows:

```
<RelativeLayout xmlns:android=
      "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginTop="27dp"
    android:layout_marginBottom="27dp"
    android:layout_marginLeft="48dp"
    android:layout_marginRight="48dp">

    <!-- Screen elements ... -->

</RelativeLayout>
```

Besides, for Android TV, developing for 1920 × 1080 pixels is suggested. With other hardware pixel sizes, Android will then automatically downscale layout elements if necessary.

Since users cannot navigate through tappable UI elements and instead use a D-pad for navigation, an alternative way to switch between UI elements is needed for Android TV. This can easily be accomplished by adding the `nextFocusUp`, `nextFocusDown`, `nextFocusLeft`, and `nextFocusRight` attributes to UI elements. The arguments are then ID specifications of navigate-to elements, as in `@+id/xyzElement`.

For TV playback components, the leanback library provides a couple of classes and concepts that come in handy as listed here:

- For a media browser, let your fragment extend `android.support.v17.leanback.app.BrowseSupportFragment`. The project builder wizard instead creates a deprecated `BrowseFragment`, but you can walk through the API documentation of `BrowseSupportFragment` to learn the new methodology.
- The actual media items to be presented inside the media browser are governed by a card view. The corresponding class to overwrite is `android.support.v17.leanback.widget.Presenter`.
- To show details of a selected media item, extend class `android.support.v17.leanback.app.DetailsSupportFragment`. The wizard creates the deprecated `DetailFragment` instead, but their usage is similar, and you can take a look at the API documentation for more details.
- For a UI element showing video playback, use one of `android.support.v17.leanback.app.PlaybackFragment` or `android.support.v17.leanback.app.VideoFragment`.
- Use class `android.media.session.MediaSession` to configure a “Now Playing” card.

- Direct rendering of a video stream onto a UI element is supported by class `android.media.tv.TvInputService`. Calling `onTune(...)` will start rendering the direct video stream.
- If your app needs a guide using several steps, for example to present a purchasing workflow to the user, you can use class `android.support.v17.leanback.app.GuidedStepSupportFragment`.
- To present an app to the first-time user in a noninteractive way, use class `android.support.v17.leanback.app.OnboardingSupportFragment`.

Recommendation Channels for Content Search

Recommendations shown to users come in two forms: as a recommendation row before Android 8.0 and as recommendation channels starting with Android 8.0 (API level 26). To not miss users, your app should serve both in a switch, as follows:

```
if (android.os.Build.VERSION.SDK_INT >=
      Build.VERSION_CODES.O) {
  // Recommendation channels API ...
} else {
  // Recommendations row API ...
}
```

For Android 8.0 and up, the Android TV home screen shows a global Play Next row at the top of the channel list and a number of channels each belonging to a certain app. A channel other than the Play Next row cannot belong to more than one app. Each app can define a *default channel*, which automatically shows up in the channel view. For all other channels an app might define, the user must approve them first before the channel gets shown on the home screen.

An app needs to have the following permissions to be able to manage channels:

```
<uses-permission android:name=
    "com.android.providers.tv.permission.READ_EPG_DATA"
    />
<uses-permission android:name=
    "com.android.providers.tv.permission.WRITE_EPG_DATA"
    />
```

So, add them to file `AndroidManifest.xml`.

In addition, inside your module's `build.gradle` file, add the following to the `dependencies` section (on one line):

```
implementation
      'com.android.support:support-tv-provider:27.1.1'
```

To create a channel, add a channel logo, possibly make it the default channel, and write the following:

```
val builder = Channel.Builder()
// Intent to execute when the app link gets tapped.
val appLink = Intent(...).toUri(Intent.URI_INTENT_SCHEME)

// You must use type `TYPE_PREVIEW`
builder.setType(TvContractCompat.Channels.TYPE_PREVIEW)
      .setDisplayName("Channel Name")
      .setAppLinkIntentUri(Uri.parse(appLink))
val channel = builder.build()
val channelUri = contentResolver.insert(
      TvContractCompat.Channels.CONTENT_URI,
      channel.toContentValues())

val channelId = ContentUris.parseId(channelUri)
// Choose one or the other
ChannelLogoUtils.storeChannelLogo(this, channelId,
      /*Uri*/ logoUri)
ChannelLogoUtils.storeChannelLogo(this, channelId,
      /*Bitmap*/ logoBitmap)

// optional, make it the default channel
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O)
    TvContractCompat.requestChannelBrowsable(this,
          channelId)
```

To update or delete a channel, you use the channel ID gathered from the channel creation step and then write the following:

```
// to update:
contentResolver.update(
      TvContractCompat.buildChannelUri(channelId),
      channel.toContentValues(), null, null)

// to delete:
contentResolver.delete(
      TvContractCompat.buildChannelUri(channelId),
      null, null)
```

To add a program, use the following:

```
val pbuilder = PreviewProgram.Builder()
// Intent to launch when a program gets selected
val progLink = Intent().toUri(Intent.URI_INTENT_SCHEME)

pbuilder.setChannelId(channelId)
    .setType(TvContractCompat.PreviewPrograms.TYPE_CLIP)
    .setTitle("Title")
    .setDescription("Program description")
    .setPosterArtUri(largePosterArtUri)
```

```
    .setIntentUri(Uri.parse(progLink))
    .setInternalProviderId(appProgramId)
val previewProgram = pbuilder.build()
val programUri = contentResolver.insert(
      TvContractCompat.PreviewPrograms.CONTENT_URI,
             previewProgram.toContentValues())
val programId = ContentUris.parseId(programUri)
```

To instead add a program to the Play Next row, you somewhat similarly use `WatchNextProgram.Builder` and write the following:

```
val wnbuilder = WatchNextProgram.Builder()
val watchNextType = TvContractCompat.
      WatchNextPrograms.WATCH_NEXT_TYPE_CONTINUE
wnbuilder.setType(
        TvContractCompat.WatchNextPrograms.TYPE_CLIP)
    .setWatchNextType(watchNextType)
    .setLastEngagementTimeUtcMillis(time)
    .setTitle("Title")
    .setDescription("Program description")
    .setPosterArtUri(largePosterArtUri)
    .setIntentUri(Uri.parse(progLink))
    .setInternalProviderId(appProgramId)
val watchNextProgram = wnbuilder.build()
val watchNextProgramUri = contentResolver
    .insert(
         TvContractCompat.WatchNextPrograms.CONTENT_URI,
        watchNextProgram.toContentValues())
val watchnextProgramId =
      ContentUris.parseId(watchNextProgramUri)
```

For `watchNextType`, you can use one of the following constants from `TvContractCompat.WatchNextPrograms`:

- `WATCH_NEXT_TYPE_CONTINUE`: The user stopped while watching content and can resume here.
- `WATCH_NEXT_TYPE_NEXT`: The next available program in a series is available.
- `WATCH_NEXT_TYPE_NEW`: The next available program in a series is newly available.
- `WATCH_NEXT_TYPE_WATCHLIST`: This is inserted by the system or the app when the user saves a program.

To update or delete a program, use the program ID you memorized from the program generation.

```
// to update:
contentResolver.update(
      TvContractCompat.
         buildPreviewProgramUri(programId),
      watchNextProgram.toContentValues(), null, null)
```

```
// to delete:
contentResolver.delete(
      TvContractCompat.
          buildPreviewProgramUri(programId),
      null, null)
```

A Recommendation Row for Content Search

For Android up to version 7.1 (API level 25), recommendations were handled by a special recommendation row. You must not use a recommendation row for any later version.

For an app to participate in the recommendation row for the pre-8.0 Android versions, we first create a new recommendation service as follows:

```
class UpdateRecommendationsService :
      IntentService("RecommendationService") {
  companion object {
      private val TAG = "UpdateRecommendationsService"
      private val MAX_RECOMMENDATIONS = 3
  }
  override fun onHandleIntent(intent: Intent?) {
      Log.d("LOG", "Updating recommendation cards")

      val recommendations:List<Movie> =
            ArrayList<Movie>()
      // TODO: fill recommendation movie list...

      var count = 0
      val notificationManager =
          getSystemService(Context.NOTIFICATION_SERVICE)
           as NotificationManager
      val notificationId = 42
      for (movie in recommendations) {
          Log.d("LOG", "Recommendation - " +
               movie.title!!)
      val builder = RecommendationBuilder(
          context = applicationContext,
          smallIcon = R.drawable.video_by_icon,
           id = count+1,
          priority = MAX_RECOMMENDATIONS - count,
          title = movie.title ?: "",
          description = "Description",
          image = getBitmapFromURL(
                movie.cardImageUrl ?:""),
          intent = buildPendingIntent(movie))
      val notification = builder.build()
      notificationManager.notify(
            notificationId, notification)
```

```
      if (++count >= MAX_RECOMMENDATIONS) {
        break
      }
    }
  }

  private fun getBitmapFromURL(src: String): Bitmap {
      val url = URL(src)
      return (url.openConnection() as HttpURLConnection).
      apply {
          doInput = true
      }.let {
          it.connect()
          BitmapFactory.decodeStream(it.inputStream)
      }
  }

  private fun buildPendingIntent(movie: Movie):
        PendingIntent {
      val detailsIntent =
          Intent(this, DetailsActivity::class.java)
      detailsIntent.putExtra("Movie", movie)

      val stackBuilder = TaskStackBuilder.create(this)
      stackBuilder.addParentStack(
            DetailsActivity::class.java)
      stackBuilder.addNextIntent(detailsIntent)
      // Ensure a unique PendingIntents, otherwise all
      // recommendations end up with the same
      // PendingIntent
      detailsIntent.action = movie.id.toString()

      return stackBuilder.getPendingIntent(
            0, PendingIntent.FLAG_UPDATE_CURRENT)
  }
}
```

The corresponding entry in `AndroidManifest.xml` reads as follows:

```
<service
    android:name=".UpdateRecommendationsService"
    android:enabled="true" />
```

`RecommendationBuilder` in the code refers to a wrapper class around a notification builder.

```
  class RecommendationBuilder(
    val id:Int = 0,
    val context:Context,
    val title:String,
    val description:String,
    var priority:Int = 0,
    val image: Bitmap,
```

```
    val smallIcon: Int = 0,
    val intent: PendingIntent,
    val extras:Bundle? = null
) {
    fun build(): Notification {
      val notification:Notification =
          NotificationCompat.BigPictureStyle(
              NotificationCompat.Builder(context)
                  .setContentTitle(title)
                  .setContentText(description)
                  .setPriority(priority)
                  .setLocalOnly(true)
                  .setOngoing(true)
                  .setColor(...)
                  .setCategory(
                      Notification.CATEGORY_RECOMMENDATION)
                  .setLargeIcon(image)
                  .setSmallIcon(smallIcon)
                  .setContentIntent(intent)
                 .setExtras(extras))
              .build()
        return notification
    }
}
```

We need it because creating and passing a notification is the way to tell the system about a recommendation.

What is left is a component that starts at system bootup and then regularly sends the recommendation. An example would use a broadcast receiver and an alarm for the periodic updates.

```
class RecommendationBootup : BroadcastReceiver() {
  companion object {
      private val TAG = "BootupActivity"
      private val INITIAL_DELAY: Long = 5000
  }

  override
  fun onReceive(context: Context, intent: Intent) {
      Log.d(TAG, "BootupActivity initiated")
      if (intent.action!!.endsWith(
            Intent.ACTION_BOOT_COMPLETED)) {
          scheduleRecommendationUpdate(context)
      }
  }

  private
  fun scheduleRecommendationUpdate(context: Context) {
      Log.d(TAG, "Scheduling recommendations update")
```

```
      val alarmManager =
            context.getSystemService(
            Context.ALARM_SERVICE) as AlarmManager
      val recommendationIntent = Intent(context,
            UpdateRecommendationsService::class.java)
      val alarmIntent =
            PendingIntent.getService(
            context, 0, recommendationIntent, 0)

      alarmManager.setInexactRepeating(
             AlarmManager.ELAPSED_REALTIME_WAKEUP,
             INITIAL_DELAY,
             AlarmManager.INTERVAL_HALF_HOUR,
             alarmIntent)
  }
}
```

Here's the corresponding entry in `AndroidManifest.xml`:

```
<receiver android:name=".RecommendationBootup"
          android:enabled="true"
           android:exported="false">
  <intent-filter>
    <action android:name=
        "android.intent.action.BOOT_COMPLETED"/>
  </intent-filter>
</receiver>
```

For this to work, you need this permission:

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

Android TV Content Search

Your Android TV app may contribute to the Android search framework. We described it in Chapter 8. In this section, we point out peculiarities for using search in a TV app.

Table 13-2 describes the search item fields important for TV search suggestions; the left column lists constant names from the `SearchManager` class. You can use them in your database as shown, or at least you must provide a mapping mechanism inside the app.

Table 13-2. TV Search Fields

Field	Description
SUGGEST_COLUMN_TEXT_1	Required. The name of your content.
SUGGEST_COLUMN_TEXT_2	A text description of your content.
SUGGEST_COLUMN_RESULT_CARD_IMAGE	An image/poster/cover for your content.
SUGGEST_COLUMN_CONTENT_TYPE	Required. The MIME type of your media.
SUGGEST_COLUMN_VIDEO_WIDTH	The width of your media.
SUGGEST_COLUMN_VIDEO_HEIGHT	The height of your media.
SUGGEST_COLUMN_PRODUCTION_YEAR	Required. The production year.
SUGGEST_COLUMN_DURATION	Required. The duration in milliseconds.

As for any other search provider, create a content provider for the search suggestions inside your app.

As soon as the user submits the search dialog to actually *perform* a search query, the search framework fires an intent with the action SEARCH, so you can write an activity with the appropriate intent filter as follows:

```
<activity
    android:name=".DetailsActivity"
    android:exported="true">

    <!-- Receives the search request. -->
    <intent-filter>
        <action android:name=
              "android.intent.action.SEARCH" />
    </intent-filter>

    <!-- Points to searchable meta data. -->
    <meta-data android:name="android.app.searchable"
        android:resource="@xml/searchable" />
</activity>
```

Android TV Games

While games development seems appealing at first because of the large display, it is important to keep these points in mind:

- TVs are always in landscape mode, so make sure your app is good at using landscape mode.
- For a multiplayer game, it is normally not possible to hide things from users, for example in a card game. You could connect TV apps to companion apps running on smartphones to remedy this.

- Your TV game should support gamepads, and it should prominently tell the users how to use them. Inside the `AndroidManifest.xml` file, you better declare `<uses-feature android:name = "android.hardware.gamepad" android:required = "false"/>`. If instead you write `required = "true"`, you make your app uninstallable for users who don't own gamepads.

Android TV Channels

The handling of live content, that is, the presentation of continuous, channel-based content, is governed by the TV Input Framework and various classes in the `com.android.tv`, `com.android.providers.tv`, and `android.media.tv` packages. It primarily addresses OEM manufacturers to serve as an aid to connect the TV system of Android to live streaming data. For details, please take a look at the API documentation of these packages or enter *Android Building TV Channels* in your favorite search engine.

Programming with Android Auto

Android Auto is for transferring the Android OS to a compatible car user interface. By 2018 dozens of car manufacturers have included or plan to include Android Auto in at least some of their models, so extending your app to include Android Auto features gives you new possibilities for distributing your app and improving the user experience. As an alternative mode of operation, an Android Auto app may also run on a smartphone or tablet somehow mounted in your car, which makes it usable for any type of car, with or without a compatible user interface.

Android Auto is currently limited to the following features of Android OS to be used in a car:

- Playing audio
- Messaging

Android Auto is for devices starting with Android 5.0 (API level 21). In addition, you must provide a file called `automotive_app_desc.xml` in the `res/xml` folder with the following content (or ones of the lines):

```
<automotiveApp>
   <uses name="media" />
   <uses name="notification" />
</automotiveApp>
```

In addition, in `AndroidManifest.xml` add the following in the `<application>` element:

```
<meta-data android:name=
      "com.google.android.gms.car.application"
    android:resource=
      "@xml/automotive_app_desc"/>
```

Developing for Android Auto

To develop apps for Android Auto, use Android Studio like with any other Android app. Make sure you are targeting API level 21 or higher and that you have the v4 support library added to the module's `build.gradle` file in the `dependencies` section (on one line).

```
implementation
      'com.android.support:support-v4:27.1.1'
```

Testing Android Auto for a Phone Screen

To test running Android Auto on your handheld, you must install the Android Auto app from Google Play. Then, inside the menu, tap Info and then tap ten or more times on the activity header (attention, no feedback!) until a toast notification appears to enable the developer mode. Now, tap the new menu item "Developer settings" and select the "Unknown sources" option. Restart Android Auto. On the device, enable USB debugging by tapping seven times on the build number in the Settings ➤ About screen. Afterward, in Settings ➤ Developer Options, enable USB debugging.

Testing Android Auto for a Car Screen

You can test an Auto app in the Desktop Head Unit (DHU) tool. This emulates a car user interface on your handheld. To install it, on the device, first enable USB debugging by tapping seven times on the build number in the Settings ➤ About screen. Afterward, in Settings ➤ Developer Options, enable USB debugging. After that, install the Android Auto app on your handheld.

In Android Studio, open the SDK Manager in the Tools menu and download and install the Android Auto Desktop Head Unit emulator. You'll find this option in Android SDK ➤ SDK Tools.

To run the DHU on Linux, you must install the following packages: `libsdl2-2.0-0`, `libsdl2-ttf-2.0-0`, `libportaudio2`, and `libpng12-0`. In Android Auto, enable the developer options as described earlier in the section "Testing Android Auto for a Phone Screen."

Unless it's already running, inside the Android Auto app's menu, select "Start head unit server." In Settings, tap "Connected cars" and make sure the "Add new cars to Android Auto" option is enabled.

Connect the handheld to the development machine via a USB cable, and with a terminal open and inside the Android SDK folder, advance to the `platform-tools` folder and issue the following command:

```
./adb forward tcp:5277 tcp:5277
```

You can now start the DHU tool by entering the following:

```
cd <sdk>/extras/google/auto
./desktop-head-unit
# or ./desktop-head-unit -i controller
# for rotary control
```

The DHU tool should now appear on your development machine's screen, as shown in Figure 13-2.

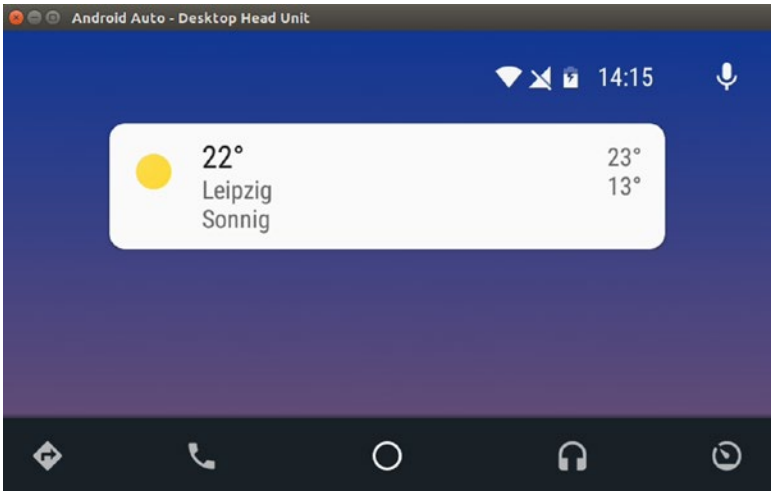

Figure 13-2. The DHU screen

Also, the last command opened a shell to the DHU tool, so commands can be entered and sent to it. Here are some interesting use cases for the shell:

- **Day and night modes**

  Enter `daynight` in the console. Click the DHU screen to give it the focus and press N on the keyboard to toggle between day and night.

- **Simulated microphone input**

  Enter `mic play /path/to/sound/file/file.wav` to send a sound file as simulated microphone input. Common voice commands can be found in `<sdk>/extras/google/auto/voice/`.

- **Sleep**

  Enter `sleep <N>` to cause the system to sleep for `N` seconds.

- **Tap**

  Enter `tap <X> <Y>` to simulate a tap event at some coordinates (useful for test scripts).

If you have engaged the rotary controller mode, entering `dpad` and any of the following will simulate a rotary control action:

- `up`, `down`, `left`, or `right`: Simulates moving. This is the same as the arrow keys.
- `soft left` or `soft right`: Simulates pressing the side buttons (only on some devices). This is the same as using Shift and the arrow keys.
- `click`: Simulates pressing the controller. This is the same as pressing Return.

- `back`: Simulates pressing the Back button (only on some devices). This is the same as pressing Backspace.
- `rotate left` or `rotate right`: Simulates a controller rotation. This is the same as pressing 1 or 2.
- `flick left` or `flick right`: Simulates a fast spin of the controller. This is the same as pressing Shift+1 or Shift+2.

Develop Audio Playback on Auto

If your app provides audio services to Android Auto, you define a media browser service in the file `AndroidManifest.xml` as follows:

```
<service android:name=".MyMediaBrowserService"
              android:exported="true">
    <intent-filter>
      <action android:name=
          "android.media.browse.MediaBrowserService"/>
    </intent-filter>
</service>
```

To additionally define a notification icon for your app in `<application>`, write the following:

```
<meta-data android:name=
      "com.google.android.gms.car.notification.SmallIcon"
    android:resource=
      "@drawable/ic_notification" />
```

We'll take care of an implementation for a media browser service soon, but first we will talk about some status inquiry methods. First, if your app needs to find out whether an Android Auto user got connected to your app, you add a broadcast receiver with the intent filter `com.google.android.gms.car.media.STATUS`. The `onReceive(...)` method of the receiver class will then get an intent with an extra value keyed by `media_connection_status`. The value of this *extra* field might, for example, read `media_connected` to indicate a connection event.

In addition, the app can find out whether it is running in *car mode* by using the following query:

```
fun isCarUiMode(c:Context):Boolean {
    val uiModeManager =
          c.getSystemService(Context.UI_MODE_SERVICE) as
          UiModeManager
    return uiModeManager.getCurrentModeType() ==
           Configuration.UI_MODE_TYPE_CAR
  }
```

Now let's get back to the media browser implementation. The most important thing to do is let the service implement the abstract class `MediaBrowserServiceCompat`. In its overwritten `onCreate(...)` method, you create and register a `MediaSessionCompat` object.

```
public void onCreate() {
  super.onCreate()
  ...
  // Start a MediaSession
  val mSession = MediaSessionCompat(
        this, "my session tag")
  val token:MediaSessionCompat.Token =
        mSession.sessionToken

  // Set a callback object to handle play
  /control requests
  mSession.setCallback(
        object : MediaSessionCompat.Callback() {
      // overwrite methods here for
      // playback controls...
  })
  ...
}
```

In this service, you must implement the following methods:

- **onGetRoot(...)**

  This is supposed to give back the top node of your content hierarchy.

- **onLoadChildren(...)**

  Here you return the children of a node inside the hierarchy.

To minimize the car driver's distraction, your app should be able to listen to voice commands. To enable voice commands like "Play XYZ or APP_NAME," just add the following to the file `AndroidManifest.xml`:

```
<activity>
  <intent-filter>
    <action android:name=
       "android.media.action.MEDIA_PLAY_FROM_SEARCH" />
    <category android:name=
       "android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

This will let the framework call the `onPlayFromSearch(...)` callback from the `MediaSessionCompat.Callback` listener you added to the session. The second parameter passed in there may as a `Bundle` contain extra search select information you can use to filter the results you want to return in your app. Use one of the `MediaStore.EXTRA_*` constants to retrieve values from that `Bundle` parameter.

To allow for playback voice control actions like “Next song” or “Resume music,” add the following as flags to the media session object:

```
mSession.setFlags(
      MediaSession.FLAG_HANDLES_MEDIA_BUTTONS or
      MediaSession.FLAG_HANDLES_TRANSPORT_CONTROLS)
```

Develop Messaging on Auto

Your Auto app may contribute to Android Auto messaging. More precisely, you can do one or more of the following:

- Post a notification to Android Auto. The notification consists of the message itself and a conversation ID, which is used to group messages. You don’t have to do that grouping yourself, but assigning a conversation ID to the notification is important, so the framework can let the user know a notification belongs to a conversation with one dedicated communication partner.
- When the user listens to the message, the framework will fire a “message read” intent that your app can catch.
- The user can send a reply using the Auto framework. This gets accompanied by another “message reply” intent fired by the framework and catchable by your app.

To catch “message read” and “message reply” events, you write receivers as indicated by the following entries in `AndroidManifest.xml`:

```
<application>
  ...
  <receiver android:name=".MyMessageReadReceiver"
            android:exported="false">
      <intent-filter>
        <action android:name=
          "com.myapp.auto.MY_ACTION_MESSAGE_READ"/>
      </intent-filter>
  </receiver>

  <receiver android:name=".MyMessageReplyReceiver"
        android:exported="false">
      <intent-filter>
        <action android:name=
          "com.myapp.auto.MY_ACTION_MESSAGE_REPLY"/>
      </intent-filter>
  </receiver>
  ...
</application>
```

You must tell Auto that you want to receive such events. You do so by preparing appropriate `PendingIntent` objects as follows:

```
val msgReadIntent = Intent().apply {
  addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES)
  setAction("com.myapp.auto.MY_ACTION_MESSAGE_READ")
  putExtra("conversation_id", thisConversationId)
  setPackage("com.myapp.auto")
}.let {
  PendingIntent.getBroadcast(applicationContext,
      thisConversationId,
      it,
      PendingIntent.FLAG_UPDATE_CURRENT)
}

val msgReplyIntent = Intent().apply {
  addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES)
  setAction("com.myapp.auto.MY_ACTION_MESSAGE_REPLY")
  putExtra("conversation_id", thisConversationId)
  setPackage("com.myapp.auto")
}.let {
  PendingIntent.getBroadcast(applicationContext,
      thisConversationId,
      it,
      PendingIntent.FLAG_UPDATE_CURRENT)
}
```

Here, `com.myapp.auto` is the package name associated with your app. You have to appropriately substitute it.

To further handle the interaction with Android Auto, we need an `UnreadConversation` object that you can generate as follows:

```
// Build a RemoteInput for receiving voice input
// in a Car Notification
val remoteInput =
    RemoteInput.Builder(MY_VOICE_REPLY_KEY)
    .setLabel("The label")
    .build()

val unreadConvBuilder =
  UnreadConversation.Builder(conversationName)
      .setReadPendingIntent(msgReadIntent)
      .setReplyAction(msgReplyIntent, remoteInput)
```

Here, `conversationName` is the name of the conversation shown to the Auto user. This could also be a comma-separated list of identifiers if the conversation is with more than one user.

The UnreadConversation builder is not ready yet. We first add the message as follows:

```
unreadConvBuilder.addMessage(messageString)
      .setLatestTimestamp(currentTimestamp)
```

Next we prepare a NotificationCompat.Builder object. To that builder we add the unreadConvBuilder builder from earlier, fetch a NotificationManager from the system, and finally send the message.

```
val notificationBuilder =
    NotificationCompat.Builder(applicationContext)
      .setSmallIcon(smallIconResourceID)
      .setLargeIcon(largeIconBitmap)

notificationBuilder.extend(CarExtender()
  .setUnreadConversation(unreadConvBuilder.build())

NotificationManagerCompat.from(/*context*/this).run {
    notify(notificationTag,
           notificationId,
            notificationBuilder.build())
```

What is left is handling the "message read" and "message reply" events, if you registered for them. For this you write corresponding BroadcastReceiver classes, as indicated by the entries in AndroidManifest.xml. Note that for the "message reply" action you need to use a certain way to get hold of the message.

```
val remoteInput =
      RemoteInput.getResultsFromIntent(intent)?.let {
           it.getCharSequence(MY_VOICE_REPLY_KEY)
      } ?: ""
```

Playing and Recording Sound

Playing sound in Android means one or two things, or both together:

- *Short sound snippets*: You typically play them as a feedback to user interface actions, such as pressing a button or entering something in an edit field. Another use case is games, where certain events could be mapped to short audio fragments. Especially for UI reactivity, make sure you don't annoy users and provide for a possibility to readily mute audio output.
- *Music playback*: You want to play music pieces with a duration longer than a few seconds.

For short audio snippets, you use a SoundPool; for music pieces, you use a MediaPlayer. We talk about them and also audio recording in the following sections.

Short Sound Snippets

For short sound snippets, you use a `SoundPool` and preload the sounds during initialization. You cannot immediately use the sound snippets after you load them using one of the `SoundPool.load(...)` methods. Instead, you have to wait until all sounds are loaded. The suggested way is not to wait for some time as you frequently can read in some blogs. Instead, listen to sound load events and count finished snippets. You can let a custom class do that, as follows:

```
class SoundLoadManager(val ctx:Context) {
  var scheduled = 0
  var loaded = 0
  val sndPool:SoundPool
  val soundPoolMap = mutableMapOf<Int,Int>()
  init {
      sndPool =
          if (Build.VERSION.SDK_INT >=
                 Build.VERSION_CODES.LOLLIPOP) {
            SoundPool.Builder()
               .setMaxStreams(4)
               .setAudioAttributes(
                  AudioAttributes.Builder()
                .setUsage(
                   AudioAttributes.USAGE_MEDIA)
                .setContentType(
                    AudioAttributes.CONTENT_TYPE_MUSIC)
                .build()
            ).build()
        } else {
            SoundPool(4,
                 AudioManager.STREAM_MUSIC,
                 100)
        }
     sndPool.setOnLoadCompleteListener({
         sndPool, sampleId, status ->
         if(status != 0) {
             Log.e("LOG",
                   "Sound could not be loaded")
         } else {
             Log.i("LOG", "Loaded sample " +
                   sampleId + ", status = " +
                   status)
         }
         loaded++
     })
  }
  fun load(resourceId:Int) {
      scheduled++
      soundPoolMap[resourceId] =
            sndPool.load(ctx, resourceId, 1)
  }
```

```
  fun allLoaded() = scheduled == loaded

  fun play(rsrcId: Int, loop: Boolean):Int {
      return soundPoolMap[rsrcId]?.run {
          val audioManager = ctx.getSystemService(
                Context.AUDIO_SERVICE) as AudioManager
          val curVolume = audioManager.
                 getStreamVolume(
                 AudioManager.STREAM_MUSIC)
          val maxVolume = audioManager.
                 getStreamMaxVolume(
                 AudioManager.STREAM_MUSIC)
          val leftVolume = 1f * curVolume / maxVolume
          val rightVolume = 1f * curVolume / maxVolume
          val priority = 1
          val noLoop = if(loop) -1 else 0
          val normalPlaybackRate = 1f
           sndPool.play(this, leftVolume, rightVolume,
                priority, noLoop, normalPlaybackRate)
      } ?: -1
  }
}
```

Note the following about this class:

- Loads and saves an instance of `SoundPool`. The constructor is deprecated, which is why we use different ways of initializing it, dependent on the Android API level. The parameters shown here may be adapted according to your needs; please see the API documentation of `SoundPool`, `SoundPool.Builder`, and `AudioAttributes.Builder`.
- Provides for a `load()` method with a resource ID as an argument. This could, for example, be a WAV file inside the `res/raw` folder.
- Provides for an `allLoaded()` method that you can use to check whether all sounds have been loaded.
- Provides for a `play()` method that you can use to play a loaded sound. This will do nothing if the sound is not loaded yet. This will return the stream ID if the sound gets actually played, or else `-1`.

To use the class, create a field with an instance. Upon initialization, for example, in an activity's `onCreate(...)` method, load the sounds and invoke `play()` to start playing.

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

    ...
```

```
    soundLoadManager = SoundLoadManager(this)
    with(soundLoadManager) {
      load(R.raw.click)
      // more ...
    }
}

fun go(v: View) {
    Log.e("LOG", "All sounds loaded = " +
         soundLoadManager.allLoaded())
    val strmId = soundLoadManager.play(
           R.raw.click, false)
    Log.e("LOG", "Stream ID = " + strmId.toString())
}
```

The `SoundPool` class also allows for stopping and resuming sounds. You can appropriately extend the `SoundLoadManager` class to take that into account if you need it.

Playing Media

The class `MediaPlayer` is all you need to register and play a music clip of arbitrary length and arbitrary origin. It is a state engine and as such not particularly easy to handle, but we first talk about permissions we might need to operate a media player.

- If your app needs to play media originating on the Internet, you must allow Internet access by adding the following to the file `AndroidManifest.xml`:

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

- If you want to prevent your playback from being interrupted by the device going asleep, you need to acquire wake locks. We will be talking more about that in a moment, but for this to be possible at all, you need to add the following permission to `AndroidManifest.xml`:

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

To see what to do further to acquire permissions in your code, please refer to Chapter 7.

With the necessary permissions set up, we can now handle the `MediaPlayer` class. As already mentioned, an instance of it creates a state machine, and the transitions from state to state correspond to various playback states. In more detail, the object can be in one of the following states:

- **Idle**

  Once constructed by the default constructor or after a `reset()`, the player is in *idle* state.

- **Initialized**

  Once the data source gets set via `setDataSource(...)`, the player is in *initialized* state. Unless you first use a `reset()`, calling `setDataSource(...)` again results in an error.

- **Prepared**

  The preparation transition prepares some resources and data streams to be used for the playback. Because it might take some time, especially for stream resources originating from data sources on the Internet, there are two possibilities to engage that transition: the `prepare()` method executes that step and blocks the program flow until it finishes, while the `prepareAsync()` method sends the preparation to the background. In the latter case, you have to register a listener via `setOnPreparedListener(...)` to find out when the preparation step actually finished. You must do the preparation before you can start after initialization, and you must do it again after a `stop()` method before you can start the playback again.

- **Started**

  After a successful preparation, the playback can be started by calling `start()`.

- **Paused**

  After a `start()`, you can temporarily suspend the playback by calling `pause`. Calling `start` again resumes the playback at the current playback position.

- **Stopped**

  You can stop the playback, either while it is running or while it is paused, by invoking `stop()`. Once stopped, it is not allowed to start again, unless the preparation step got repeated first.

- **Completed**

  Once the playback is completed and no looping is active, the *completed* state gets entered. You can either stop from here or start again.

Note that the various static `create(...)` factory methods gather several transitions. For details, please see the API documentation.

To give you an example, a basic player UI interface for playing a music file from inside the `assets` folder, utilizing a synchronous preparation and with a start/pause button and a stop button, looks like this:

```
var mPlayer: MediaPlayer? = null
fun btnText(playing:Boolean) {
    startBtn.text = if(playing) "Pause" else "Play"
}
fun goStart(v:View) {
    mPlayer = mPlayer?.run {
        btnText(!isPlaying)
        if(isPlaying)
            pause()
```

```
        else
            start()
        this
    } ?: MediaPlayer().apply {
        setOnCompletionListener {
            btnText(false)
        release()
        mPlayer = null
    }
    val fd: AssetFileDescriptor =
          assets.openFd("tune1.mp3")
    setDataSource(fd.fileDescriptor)
    prepare() // synchronous
    start()
    btnText(true)
  }
}

fun goStop(v:View) {
    mPlayer?.run {
        stop()
        prepare()
        btnText(false)
    }
}
```

The code is mostly self-explanatory. The `goStart()` and `goStop()` methods get called once the buttons get pressed, and `btnText(...)` is used to indicate state changes. The construct used here might look strange first, but all it does is: if the `mPlayer` object is not `null`, do `(A)` and finally perform a void assignment to itself. Otherwise, construct it, and then apply `(B)` to it.

```
mPlayer = mPlayer?.run {
    (A)
    this
} ?: MediaPlayer().apply {
    (B)
}
```

For that example to work, you must have buttons with IDs `startBtn` and `stopBtn` in your layout, connect them via `android:onclick="goStop"` and `android:onclick="goStart"`, and have a file called `tune1.mp3` inside your `assets/` folder. The example switches the button text between the “Play” and “Pause” labels; you could of course instead use `ImageButton` views here and change icons once pressed.

To use any other data source, including online streams from the Internet, apply one of the various `setDataSource(...)` alternatives or use one of the static `create(...)` methods. To monitor the various state transitions, add appropriate listeners via `setOn...Listener(...)`. It is further suggested to immediately call `release()` on a `MediaPlayer` object once you are done with it to free no longer used system resources.

The playback of some music can also be handled in the background, for example using a service instead of an activity. In such a case, if you want to avoid the device interrupting a playback because it decides to go into a sleep mode, you acquire wake locks as follows to avoid the CPU going to sleep:

```
mPlayer.setWakeMode(applicationContext,
      PowerManager.PARTIAL_WAKE_LOCK)
```

This avoids the network connection being interrupted:

```
val wifiLock = (applicationContext.getSystemService(
 Context.WIFI_SERVICE) as WifiManager)
    .createWifiLock(WifiManager.WIFI_MODE_FULL, "
 myWifilock")
.run {
  acquire()
  this
}
... later:
wifiLock.release()
```

Recording Audio

For recording audio, you use the class `MediaRecorder`. Using it is rather straightforward, as shown here:

```
val mRecorder = MediaRecorder().apply {
   setAudioSource(MediaRecorder.AudioSource.MIC)
   setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP)
   setOutputFile(mFileName)
   setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB)
}
mRecorder.prepare()
mRecorder.start()

... later:
mRecorder.stop()
```

For other options regarding input, media format, and output, please see the API documentation of class `MediaRecorder`.

Using the Camera

An application showing things to the user always has been a predominant application area of computers. First it was text, later pictures, and even later movies. Only during the last decades has the opposite, letting the user show things, gained considerable attention. With handhelds being equipped with cameras of increasing quality, the need for apps that are able to handle camera data has come up. Android helps a lot here; an app can tell

the Android OS to take a picture or record a movie and save it somewhere, or it can take complete control over the camera hardware and continuously monitor camera data and change zoom, exposure, and focus on demand.

We will be talking about all that in the following sections. If you need features or settings that aren't described here, the API documentation serves as a starting point for extended research.

Taking a Picture

A high-level approach to communicate with the camera hardware is the IT counterpart of this order: "Take a picture and save it somewhere I tell you." To accomplish that, assuming the handheld actually has a camera and you have the permission to use it, you call a certain intent telling the path name where to save the image. Upon intent result retrieval, you have access to the image data, both directly to a low-resolution thumbnail and to the full image data at the place requested.

We start by telling Android that our app needs a camera. This happens via an `<uses-feature>` element inside the file `AndroidManifest.xml`.

```
<uses-feature android:name="android.hardware.camera"
              android:required="true" />
```

Inside your app, you will then do a runtime check and act accordingly.

```
if (!packageManager.hasSystemFeature(
      PackageManager.FEATURE_CAMERA)) {
    ...
 }
```

To declare the permissions necessary, you write inside the manifest file `AndroidManifrest.xml` in the `<manifest>` element.

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

To check that permission and in case acquire it, see Chapter 7. If you want to save the picture to a publicly available store so other apps can see it, you additionally need the permission `android.permission.WRITE_EXTERNAL_STORAGE` declared and acquired the same way. To instead save the picture data to a space private to the app, you declare a slightly different permission, as shown here:

```
<uses-permission android:name=
      "android.permission.WRITE_EXTERNAL_STORAGE"
      android:maxSdkVersion="18"/>
```

This declaration is necessary only up to Android 4.4 (API level 18).

We need to do some extra work to access the image data storage. Apart from the permission we just described, we need access to the storage on a content provider security level. This means, inside the `<application>` element of `AndroidManifest.xml`, add the following:

```
<provider
      android:name=
            "android.support.v4.content.FileProvider"
      android:authorities=
            "com.example.autho.fileprovider"
      android:exported="false"
      android:grantUriPermissions="true">
      <meta-data
          android:name=
              "android.support.FILE_PROVIDER_PATHS"
          android:resource="@xml/file_paths">
      </meta-data>
</provider>
```

Inside a file `res/xml/file_paths.xml`, write the following:

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android=
      "http://schemas.android.com/apk/res/android">
  <external-path name="my_images" path=
      "Android/data/com.example.pckg.name/files/Pictures"
  />
</paths>
```

The value inside the path attribute depends on whether we save the pictures in the publicly available storage or in the app's private data space.

- Use `Android/data/com.example.package.name/files/Pictures` if you want to save the image to the app's private data space.
- Use `Pictures` if you want to save the image to the public data space.

> **Note** If you use the app's private data space, all pictures will be deleted if the app gets uninstalled.

To start the system's camera, first create an empty file to write the picture taken and then create and fire an intent as follows:

```
val REQUEST_TAKE_PHOTO = 42
var photoFile:File? = null
fun dispatchTakePictureIntent() {
  fun createImageFile():File {
    val timeStamp =
          SimpleDateFormat("yyyyMMdd_HHmmss_SSS",
```

```
          Locale.US).format(Date())
    val imageFileName = "JPEG_" + timeStamp + "_"

    val storageDir =
          Environment.getExternalStoragePublicDirectory(
          Environment.DIRECTORY_PICTURES)
    // To instead take the App's private space:
    // val storageDir =
    // getExternalFilesDir(
    // Environment.DIRECTORY_PICTURES)
    val image = File.createTempFile(
          imageFileName,
          ".jpg",
          storageDir)
          return image
}

val takePictureIntent =
      Intent(MediaStore.ACTION_IMAGE_CAPTURE)
val canHandleIntent = takePictureIntent.
      resolveActivity(packageManager) != null
if (canHandleIntent) {
      photoFile = createImageFile()
      Log.e("LOG","Photo output File: ${photoFile}")
      val photoURI = FileProvider.getUriForFile(this,
            "com.example.autho.fileprovider",
            photoFile!!)
      Log.e("LOG","Photo output URI: ${photoURI}")
      takePictureIntent.putExtra(
            MediaStore.EXTRA_OUTPUT, photoURI)
      startActivityForResult(takePictureIntent,
            REQUEST_TAKE_PHOTO)
    }
}
dispatchTakePictureIntent()
```

Note that the second parameter in `FileProvider.getUriForFile()` designates the authority and as such must also show up in the file `AndroidManifest.xml` inside the `<provider>` element, as shown earlier.

After the photo has been taken, the app's `onActivityResult()` can be used to fetch the image data.

```
override
fun onActivityResult(requestCode: Int, resultCode: Int,
      data: Intent) {
  if ((requestCode and 0xFFFF) == REQUEST_TAKE_PHOTO
        && resultCode == Activity.RESULT_OK) {
    val bmOptions = BitmapFactory.Options()
    BitmapFactory.decodeFile(
```

```
          photoFile?.getAbsolutePath(), bmOptions)?.run {
      imgView.setImageBitmap(this)
    }
  }
}
```

Here, `imgView` points to an `ImageView` element inside the UI layout.

> **Caution** Although it is implied in the API documentation, the returned intent does not reliably contain a thumbnail image in its `data` field. Some devices do that, but others do not.

Since we use the `photoFile` field to transport the image file's name, we must take care that it can survive activity restarts. To make sure it gets persisted, write the following:

```
override
fun onSaveInstanceState(outState: Bundle?) {
  super.onSaveInstanceState(outState)
  photoFile?.run{
    outState?.putString("imgFile", absolutePath)
  }
}
```

and inside `onCreate(...)` add:

```
savedInstanceState?.run {
  photoFile = getString("imgFile")?.let {File(it)}
}
```

Only if you used publicly available space to store the picture can you advertise the image to the system's media scanner. Do so by writing the following:

```
val mediaScanIntent =
      Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE)
val contentUri = Uri.fromFile(photoFile)
mediaScanIntent.setData(contentUri)
sendBroadcast(mediaScanIntent)
```

Recording a Video

Recording a video using the system's app does not substantially differ from taking a picture, as described in the preceding section. The rest of this section assumes you worked through that section already.

First, we need a different entry inside file `res/xml/file_paths.xml`. Since we are now addressing the video section, write the following:

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android=
           "http://schemas.android.com/apk/res/android">
```

```
    <external-path name="my_videos"
                   path="Android/data/de.pspaeth.camera/
   files/Movies" />
</paths>
```

To save videos in the app's private data space or to instead use the public data space available to all apps, use this:

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android=
           "http://schemas.android.com/apk/res/android">
    <external-path name="my_videos"
            path="Movies" />
</paths>
```

Then, to tell the Android OS to start recording a video and save the data to a file of our choice, write the following:

```
var videoFile:File? = null
val REQUEST_VIDEO_CAPTURE = 43

fun dispatchRecordVideoIntent() {
  fun createVideoFile(): File {
    val timeStamp =
          SimpleDateFormat("yyyyMMdd_HHmmss_SSS",
          Locale.US).format(Date())
    val imageFileName = "MP4_" + timeStamp + "_"
    val storageDir =
          Environment.getExternalStoragePublicDirectory(
          Environment.DIRECTORY_MOVIES)
    // To instead tke the App's private space:
    // val storageDir = getExternalFilesDir(
    // Environment.DIRECTORY_MOVIES)
    val image = File.createTempFile(
          imageFileName,
          ".mp4",
          storageDir)
    return image
  }

  val takeVideoIntent =
        Intent(MediaStore.ACTION_VIDEO_CAPTURE)
  if (takeVideoIntent.resolveActivity(packageManager)
        != null) {
    videoFile = createVideoFile()
    val videoURI = FileProvider.getUriForFile(this,
          "com.example.autho.fileprovider",
          videoFile!!)
    Log.e("LOG","Video output URI: ${videoURI}")
    takeVideoIntent.putExtra(MediaStore.EXTRA_OUTPUT,
          videoURI)
```

```
    startActivityForResult(
          takeVideoIntent, REQUEST_VIDEO_CAPTURE)
    }
}
dispatchRecordVideoIntent()
```

To eventually fetch the video data after the recording completes, add the following to `onActivityResult(...)`:

```
if((requestCode == REQUEST_VIDEO_CAPTURE and 0xFFFF) &&
      resultCode == Activity.RESULT_OK) {
    videoView.setVideoPath(videoFile!!.absolutePath)
    videoView.start()
}
```

Here, `videoView` points to a `VideoView` inside your layout file.

Also, since we need to make sure the `videoFile` member survives an activity restart, add it to `onSaveInstanceState(...)` and `onCreate()` as shown earlier for the `photoFile` field.

Writing Your Own Camera App

Using intents to tell the Android OS to take a picture for us or record a video might be fine for many use cases. But as soon as you need to have more control over the camera or the GUI, you need to write your own camera access code using the camera API. In this section, I will show you an app that can do both—show you a preview and let you take a still image.

> **Note** In this book, we mostly allow for Android versions 4.1 or higher. In the current section, we deviate a little from this policy. The deprecated camera API before API level 21 (Android 5.0) differs a lot from the new camera API since level 21. That is why here we choose to use the new API, which by mid-2018 addresses 85 percent or more of all devices. For the old API, please see the online documentation.

We start with three utility classes. The first class is an extension of a `TextureView`. We use a `TextureView` since it allows for a more rapid connection between the camera hardware and the screen, and we extend it so it gets adapted better to the fixed ratio output of the camera. The listing reads as follows:

```
/**
 * A custom TextureView which is able to automatically
 * crop its size according to an aspect ratio set
 */
class AutoFitTextureView : TextureView {
    constructor(context: Context) : super(context)
    constructor(context: Context, attrs: AttributeSet?) :
         super(context, attrs)
    constructor(context: Context, attrs: AttributeSet?,
```

```
        attributeSetId: Int) :
        super(context, attrs, attributeSetId)

    var mRatioWidth = 0
    var mRatioHeight = 0

    /**
     * Sets the aspect ratio for this view. The size of
     * the view will be measured based on the ratio
     * calculated from the parameters. Note that the
     * actual sizes of parameters don't matter, that
     * is, calling setAspectRatio(2, 3) and
     * setAspectRatio(4, 6) make the same result.
     *
     * @param width Relative horizontal size
     * @param height Relative vertical size
     */
    fun setAspectRatio(width:Int, height:Int) {
        if (width < 0 || height < 0) {
            throw IllegalArgumentException(
                  "Size cannot be negative.");
        }
        mRatioWidth = width;
        mRatioHeight = height;
        requestLayout()
    }

    override
    fun onMeasure(widthMeasureSpec:Int,
          heightMeasureSpec:Int) {
        super.onMeasure(
              widthMeasureSpec, heightMeasureSpec)
        val width = MeasureSpec.getSize(widthMeasureSpec)
        val height = MeasureSpec.getSize(heightMeasureSpec)
        if (0 == mRatioWidth || 0 == mRatioHeight) {
            setMeasuredDimension(width, height)
        } else {
            val ratio = 1.0 * mRatioWidth / mRatioHeight
            if (width < height * ratio) {
            setMeasuredDimension(
                  width, (width / ratio).toInt())
        } else {
            setMeasuredDimension(
                 (height * ratio).toInt(), height)
        }
    }
  }
}
```

The next utility class queries the system for a back camera and once found stores its characteristics. It reads as follows:

```
/**
 * Find a backface camera
 */
class BackfaceCamera(context:Context) {
  var cameraId: String? = null
  var characteristics: CameraCharacteristics? = null

  init {
      val manager = context.getSystemService(
            Context.CAMERA_SERVICE) as CameraManager
      try {
            manager.cameraIdList.find {
                manager.getCameraCharacteristics(it).
                get(CameraCharacteristics.LENS_FACING) ==
                    CameraCharacteristics.LENS_FACING_BACK
            }.run {
                cameraId = this
                characteristics = manager.
                      getCameraCharacteristics(this)
            }
        } catch (e: CameraAccessException) {
            Log.e("LOG", "Cannot access camera", e)
        }
  }
}
```

The third utility class performs a couple of calculations that help us to appropriately map camera output dimensions to the texture view size. It reads as follows:

```
/**
 * Calculates and holds preview dimensions
 */
class PreviewDimension {

  companion object {
      val LOG_KEY = "PreviewDimension"

      // Max preview width guaranteed by Camera2 API
      val MAX_PREVIEW_WIDTH = 1920

      // Max preview height guaranteed by Camera2 API
      val MAX_PREVIEW_HEIGHT = 1080

      val ORIENTATIONS = SparseIntArray().apply {
          append(Surface.ROTATION_0, 90);
          append(Surface.ROTATION_90, 0);
          append(Surface.ROTATION_180, 270);
          append(Surface.ROTATION_270, 180);
}
```

As a companion function, we need a method that, given sizes supported by a camera, chooses the smallest one that is at least as large as the respective texture view size, that is at most as large as the respective max size, and whose aspect ratio matches the specified value. If such a size doesn't exist, it chooses the largest one that is at most as large as the respective max size and whose aspect ratio matches with the specified value.

```
/**
 * Calculate the optimal size.
 *
 * @param choices           The list of sizes
 * that the camera supports for the intended
 * output class
 * @param textureViewWidth  The width of the
 * texture view relative to sensor coordinate
 * @param textureViewHeight The height of the
 * texture view relative to sensor coordinate
 * @param maxWidth          The maximum width
 * that can be chosen
 * @param maxHeight         The maximum height
 * that can be chosen
 * @param aspectRatio       The aspect ratio
 * @return The optimal size, or an arbitrary one
 * if none were big enough
 */
fun chooseOptimalSize(choices: Array<Size>?,
      textureViewWidth: Int,
      textureViewHeight: Int,
      maxWidth: Int, maxHeight: Int,
      aspectRatio: Size): Size {

    // Collect the supported resolutions that are
    // at least as big as the preview Surface
    val bigEnough = ArrayList<Size>()
    // Collect the supported resolutions that are
    // smaller than the preview Surface
    val notBigEnough = ArrayList<Size>()
    val w = aspectRatio.width
    val h = aspectRatio.height
    choices?.forEach { option ->
      if (option.width <= maxWidth &&
            option.height <= maxHeight &&
            option.height ==
                  option.width * h / w) {
          if (option.width >= textureViewWidth
              && option.height >=
                 textureViewHeight) {
              bigEnough.add(option)
          } else {
              notBigEnough.add(option)
          }
      }
    }
```

```
    // Pick the smallest of those big enough. If
    // there is no one big enough, pick the
    // largest of those not big enough.
    if (bigEnough.size > 0) {
        return Collections.min(bigEnough,
              CompareSizesByArea())
    } else if (notBigEnough.size > 0) {
        return Collections.max(notBigEnough,
              CompareSizesByArea())
    } else {
        Log.e(LOG_KEY,
              "Couldn't find any suitable size")
                 return Size(textureViewWidth,
                       textureViewHeight)
            }
        }

       /**
         * Compares two sizes based on their areas.
         */
        class CompareSizesByArea : Comparator<Size> {
            override
            fun compare(lhs: Size, rhs: Size): Int {
                // We cast here to ensure the
                // multiplications won't overflow
                return Long.signum(lhs.width.toLong() *
                      lhs.height -
                      rhs.width.toLong() * rhs.height)
            }
        }
    }

    internal var rotatedPreviewWidth: Int = 0
    internal var rotatedPreviewHeight: Int = 0
    internal var maxPreviewWidth: Int = 0
    internal var maxPreviewHeight: Int = 0
    internal var sensorOrientation: Int = 0
    internal var previewSize: Size? = null
```

We need a method that calculates the preview dimension, including the sensor orientation. The method `calcPreviewDimension()` does exactly that.

```
fun calcPreviewDimension(width: Int, height: Int,
      activity: Activity, bc: BackfaceCamera) {
    // Find out if we need to swap dimension to get
    // the preview size relative to sensor coordinate.
    val displayRotation =
        activity.windowManager.defaultDisplay.rotation

    sensorOrientation = bc.characteristics!!.
        get(CameraCharacteristics.SENSOR_ORIENTATION)
    var swappedDimensions = false
```

```
    when (displayRotation) {
        Surface.ROTATION_0, Surface.ROTATION_180 ->
            if (sensorOrientation == 90 ||
                sensorOrientation == 270) {
            swappedDimensions = true
        }
        Surface.ROTATION_90, Surface.ROTATION_270 ->
            if (sensorOrientation == 0 ||
                sensorOrientation == 180) {
            swappedDimensions = true
        }
        else -> Log.e("LOG",
            "Display rotation is invalid: " +
            displayRotation)
    }

    val displaySize = Point()
    activity.windowManager.defaultDisplay.
        getSize(displaySize)
    rotatedPreviewWidth = width
    rotatedPreviewHeight = height
    maxPreviewWidth = displaySize.x
    maxPreviewHeight = displaySize.y

    if (swappedDimensions) {
        rotatedPreviewWidth = height
        rotatedPreviewHeight = width
        maxPreviewWidth = displaySize.y
        maxPreviewHeight = displaySize.x
    }

    if (maxPreviewWidth > MAX_PREVIEW_WIDTH) {
        maxPreviewWidth = MAX_PREVIEW_WIDTH
    }

    if (maxPreviewHeight > MAX_PREVIEW_HEIGHT) {
        maxPreviewHeight = MAX_PREVIEW_HEIGHT
    }
}

/**
 * Retrieves the JPEG orientation from the specified
 * screen rotation.
 *
 * @param rotation The screen rotation.
 * @return The JPEG orientation
 *       (one of 0, 90, 270, and 360)
 */
fun getOrientation(rotation: Int): Int {
    // Sensor orientation is 90 for most devices, or
    // 270 for some devices (eg. Nexus 5X). We have
    // to take that into account and rotate JPEG
```

```
    // properly. For devices with orientation of 90,
    // we simply return our mapping from ORIENTATIONS.
    // For devices with orientation of 270, we need
    // to rotate the JPEG 180 degrees.
    return (ORIENTATIONS.get(rotation) +
          sensorOrientation + 270) % 360
}
```

To allow for a correct preview image presentation, we use the method `getTransformationMatrix()`, as shown here:

```
fun getTransformationMatrix(activity: Activity,
      viewWidth: Int, viewHeight: Int): Matrix {
    val matrix = Matrix()
    val rotation = activity.windowManager.
          defaultDisplay.rotation
    val viewRect = RectF(0f, 0f,
          viewWidth.toFloat(), viewHeight.toFloat())
    val bufferRect = RectF(0f, 0f,
          previewSize!!.height.toFloat(),
          previewSize!!.width.toFloat())
    val centerX = viewRect.centerX()
    val centerY = viewRect.centerY()
    if (Surface.ROTATION_90 == rotation
          || Surface.ROTATION_270 == rotation) {
        bufferRect.offset(
              centerX - bufferRect.centerX(),
              centerY - bufferRect.centerY())
        matrix.setRectToRect(viewRect, bufferRect,
              Matrix.ScaleToFit.FILL)
        val scale = Math.max(
          viewHeight.toFloat() / previewSize!!.height,
          viewWidth.toFloat() / previewSize!!.width)
        matrix.postScale(
              scale, scale, centerX, centerY)
        matrix.postRotate(
             (90 * (rotation - 2)).toFloat(),
             centerX, centerY)
    } else if (Surface.ROTATION_180 == rotation) {
        matrix.postRotate(180f, centerX, centerY)
    }
    return matrix
  }
}
```

As in the preceding sections, we need to make sure we can acquire the necessary permissions. For this aim, add the following inside `AndroidManifest.xml`:

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

Next we write an activity, which checks and possibly acquires the permissions necessary, opens a `Camera` object whose class we will define in a moment, adds a still image capture button and a captured still image consumer callback, and takes care of a transformation matrix to have the `TextureView` object show the correctly sized preview picture. It will look like this:

```
class MainActivity : AppCompatActivity() {
  companion object {
      val LOG_KEY = "main"
      val PERM_REQUEST_CAMERA = 642
  }

  lateinit var previewDim:PreviewDimension
  lateinit var camera:Camera

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

      val permission1 =
            ContextCompat.checkSelfPermission(
            this, Manifest.permission.CAMERA)
      if (permission1 !=
            PackageManager.PERMISSION_GRANTED) {
          ActivityCompat.requestPermissions(this,
                arrayOf(Manifest.permission.CAMERA),
                PERM_REQUEST_CAMERA)
  }else{
      start()
  }
}

override fun onDestroy() {
    super.onDestroy()
    camera.close()
}

fun go(v: View) {
    camera.takePicture()
}
```

The method `start()` is used to correctly handle the camera object and set up the preview canvas. Note that when the screen is turned off and turned back on, `SurfaceTexture` is already available, and `onSurfaceTextureAvailable` will not be called. In that case, we can open a camera and start a preview from here. Otherwise, we wait until the surface is ready in `SurfaceTextureListener`.

```
private fun start() {
  previewDim = PreviewDimension()
  camera = Camera(
```

```
        this, previewDim, cameraTexture).apply {
      addPreviewSizeListener { w,h ->
          Log.e(LOG_KEY,
              "Preview size by PreviewSizeListener:
              ${w} ${h}")
          cameraTexture.setAspectRatio(w,h)
      }
      addStillImageConsumer(::dataArrived)
  }

  // Correctly handle the screen turned off and
  // turned back on.
  if (cameraTexture.isAvailable()) {
      camera.openCamera(cameraTexture.width,
            cameraTexture.height)
      configureTransform(cameraTexture.width,
            cameraTexture.height)
  } else {
      cameraTexture.surfaceTextureListener = object :
            TextureView.SurfaceTextureListener {
          override
          fun onSurfaceTextureSizeChanged(
                surface: SurfaceTexture?,
                width: Int, height: Int) {
              configureTransform(width, height)
          }
          override
          fun onSurfaceTextureUpdated(
                surface: SurfaceTexture?) {
          }
          override
          fun onSurfaceTextureDestroyed(
                surface: SurfaceTexture?): Boolean {
              return true
          }
          override
          fun onSurfaceTextureAvailable(
                surface: SurfaceTexture?,
                 width: Int, height: Int) {
              camera.openCamera(width, height)
              configureTransform(width, height)
          }
      }
  }
}

private fun dataArrived(it: ByteArray) {
    Log.e(LOG_KEY, "Data arrived: " + it.size)
    // do more with the picture...
}
```

```
private fun configureTransform(
      viewWidth: Int, viewHeight: Int) {
    val matrix =
          previewDim.getTransformationMatrix(
          this, viewWidth, viewHeight)
    cameraTexture.setTransform(matrix)
}
```

The `onRequestPermissionsResult()` callback is used to start the preview after the permission check returns from the corresponding system call.

```
override
fun onRequestPermissionsResult(requestCode: Int,
      permissions: Array<out String>,
      grantResults: IntArray) {
    super.onRequestPermissionsResult(requestCode,
          permissions, grantResults)
    when (requestCode) {
        PERM_REQUEST_CAMERA -> {
            if(grantResults[0] ==
                 PackageManager.PERMISSION_GRANTED) {
               start()
            }
        }
    }
  }
}
```

A corresponding layout file with a "take picture" button and the custom `ciTextureView` UI element reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  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"
  android:orientation="vertical">

  <Button
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="Go"
      android:onClick="go"/>
  <de.pspaeth.camera2.AutoFitTextureView
      android:id="@+id/cameraTexture"
      android:layout_width="400dp"
```

```
      android:layout_height="200dp"
      android:layout_marginTop="8dp"
      />
</LinearLayout>
```

Here, instead of `de.pspaeth.camera2.AutoFitTextureView`, you have to use your own class's fully qualified path.

The `Camera` class makes sure we put important activities into the background, prepares a space where the still image capture data can be put, and builds a camera session object. It also takes care of a couple sizing issues.

```
/**
 * A camera with a preview sent to a TextureView
 */
class Camera(val activity: Activity,
      val previewDim:PreviewDimension,
      val textureView:TextureView) {
  companion object {
      val LOG_KEY = "camera"
      val STILL_IMAGE_FORMAT = ImageFormat.JPEG
      val STILL_IMAGE_MIN_WIDTH = 480
      val STILL_IMAGE_MIN_HEIGHT = 480
  }

  private val previewSizeListeners =
        mutableListOf<(Int,Int) -> Unit>()
  fun addPreviewSizeListener(
        l: (Int,Int) -> Unit ) {
      previewSizeListeners.add(l)
  }

  private val stillImageConsumers =
        mutableListOf<(ByteArray) -> Unit>()
  fun addStillImageConsumer(
        l: (ByteArray) -> Unit) {
      stillImageConsumers.add(l)
  }

/**
 * An additional thread and handler for running
 * tasks that shouldn't block the UI.
 */
private var mBackgroundThread: HandlerThread? = null
private var mBackgroundHandler: Handler? = null

private var cameraDevice: CameraDevice? = null
private val backfaceCamera =
      BackfaceCamera(activity)
      // Holds the backface camera's ID

/**
```

```
 * A [Semaphore] to prevent the app from exiting
 * before closing the camera.
 */
private val cameraOpenCloseLock = Semaphore(1)

private var imageReader:ImageReader? = null

private var paused = false

private var flashSupported = false

private var activeArraySize: Rect? = null

private var cameraSession:CameraSession? = null

private var stillImageBytes:ByteArray? = null
```

The openCamera() method checks for permissions, connects to the camera data output, and initiates the connection to the camera.

```
fun openCamera(width: Int, height: Int) {
    startBackgroundThread()

    val permission1 =
          ContextCompat.checkSelfPermission(
          activity, Manifest.permission.CAMERA)
    if (permission1 !=
          PackageManager.PERMISSION_GRANTED) {
        Log.e(LOG_KEY,
              "Internal error: "+
              "Camera permission missing")
    }

    setUpCameraOutputs(width, height)
    val manager = activity.getSystemService(
          Context.CAMERA_SERVICE)
          as CameraManager
    try {
        if (!cameraOpenCloseLock.tryAcquire(
              2500, TimeUnit.MILLISECONDS)) {
            throw RuntimeException(
                  "Time out waiting.")
        }
        val mStateCallback = object :
              CameraDevice.StateCallback() {
            override
            fun onOpened(cameraDev: CameraDevice) {
                // This method is called when the
                // camera is opened.  We start camera
                // preview here.
```

```
                cameraOpenCloseLock.release()
                cameraDevice = cameraDev
                createCameraSession()
            }

            override
            fun onDisconnected(
                  cameraDev: CameraDevice) {
                cameraOpenCloseLock.release()
                cameraDevice?.close()
                cameraDevice = null
            }

            override
            fun onError(cameraDev: CameraDevice,
                  error: Int) {
                Log.e(LOG_KEY,
                      "Camera on error callback: "
                      + error);
                cameraOpenCloseLock.release()
                cameraDevice?.close()
                cameraDevice = null
            }
        }

        manager.openCamera(
              backfaceCamera.cameraId,
              mStateCallback,
              mBackgroundHandler)
    } catch (e: CameraAccessException) {
        Log.e(LOG_KEY,"Could not access camera", e)
    } catch (e: InterruptedException) {
        Log.e(LOG_KEY,
              "Interrupted while camera opening.", e)
    }
}

/**
 * Initiate a still image capture.
 */
fun takePicture() {
    cameraSession?.takePicture()
}

fun close() {
    stopBackgroundThread()
    cameraSession?.run {
        close()
    }
```

```
    imageReader?.run {
        surface.release()
        close()
        imageReader = null
    }
}
```

The following are a couple of private methods to handle the background threads and the camera session:

```
////////////////////////////////////////////////////
////////////////////////////////////////////////////

/**
 * Starts a background thread and its [Handler].
 */
private fun startBackgroundThread() {
    mBackgroundThread =
          HandlerThread("CameraBackground")
    mBackgroundThread?.start()
    mBackgroundHandler = Handler(
          mBackgroundThread!!.getLooper())
}

/**
 * Stops the background thread and its [Handler].
 */
private fun stopBackgroundThread() {
    mBackgroundThread?.run {
        quitSafely()

        try {
            join()
            mBackgroundThread = null
            mBackgroundHandler = null
        } catch (e: InterruptedException) {
        }
    }
}

private fun createCameraSession() {
    cameraSession = CameraSession(mBackgroundHandler!!,
            cameraOpenCloseLock,
            backfaceCamera.characteristics,
            textureView,
            imageReader!!,
            cameraDevice!!,
            previewDim,
            activity.windowManager.defaultDisplay.
                  rotation,
            activeArraySize!!,
            1.0).apply {
```

```
        createCameraSession()
        addStillImageTakenConsumer {
            //Log.e(LOG_KEY, "!!! PICTURE TAKEN!!!")
            for (cons in stillImageConsumers) {
                mBackgroundHandler?.post(
                    Runnable {
                      stillImageBytes?.run{
                        cons(this)
                      }
                    })
            }
        }
    }
}
```

The `setUpCameraOutputs()` method performs the hard work of connecting to the camera data output.

```
/**
 * Sets up member variables related to camera:
 * activeArraySize, imageReader, previewDim,
 * flashSupported
 *
 * @param width    The width of available size for
 *       camera preview
 * @param height The height of available size for
 *       camera preview
 */
private fun setUpCameraOutputs(
      width: Int, height: Int) {
    activeArraySize = backfaceCamera.
          characteristics?.
          get(CameraCharacteristics.
          SENSOR_INFO_ACTIVE_ARRAY_SIZE)

    val map =
        backfaceCamera.characteristics!!.get(
            CameraCharacteristics.
            SCALER_STREAM_CONFIGURATION_MAP)

    val stillSize = calcStillImageSize(map)
    imageReader =
          ImageReader.newInstance(
                stillSize.width,
                stillSize.height,
          STILL_IMAGE_FORMAT, 3).apply {
        setOnImageAvailableListener(
            ImageReader.OnImageAvailableListener {
            reader ->
                if (paused)
                  return@OnImageAvailableListener
                val img = reader.acquireNextImage()
```

```
                val buffer = img.planes[0].buffer
                stillImageBytes =
                  ByteArray(buffer.remaining())
                buffer.get(stillImageBytes)
                img.close()
            }, mBackgroundHandler)
    }

    previewDim.calcPreviewDimension(width, height,
          activity, backfaceCamera)

    val texOutputSizes =
          map?.getOutputSizes(
          SurfaceTexture::class.java)
    val optimalSize =
          PreviewDimension.chooseOptimalSize(
            texOutputSizes,
            previewDim.rotatedPreviewWidth,
            previewDim.rotatedPreviewHeight,
            previewDim.maxPreviewWidth,
            previewDim.maxPreviewHeight,
            stillSize)
    previewDim.previewSize = optimalSize

    // We fit the aspect ratio of TextureView
    // to the size of preview we picked.
    val orientation =
          activity.resources.configuration.
          orientation
    if (orientation ==
          Configuration.ORIENTATION_LANDSCAPE) {
        previewSizeListeners.forEach{
              it(optimalSize.width,
                 optimalSize.height) }
    } else {
        previewSizeListeners.forEach{
              it(optimalSize.height,
                 optimalSize.width) }
    }

    // Check if the flash is supported.
    val available =
          backfaceCamera.characteristics?.
          get(CameraCharacteristics.
                FLASH_INFO_AVAILABLE)
    flashSupported = available ?: false
}
```

One last private method calculates the still image size. This plays a role once the trigger gets pressed or a trigger press gets simulated.

```
private fun calcStillImageSize(
      map: StreamConfigurationMap): Size {
    // For still image captures, we use the smallest
    // one at least some width x height
    val jpegSizes =
              map.getOutputSizes(ImageFormat.JPEG)
        var stillSize: Size? = null
        for (s in jpegSizes) {
           if (s.height >= STILL_IMAGE_MIN_HEIGHT
                 && s.width >= STILL_IMAGE_MIN_WIDTH) {
               if (stillSize == null) {
                   stillSize = s
               } else {
                   val f =
                         (s.width * s.height).toFloat()
                   val still =
                         (stillSize.width *
                          stillSize.height).toFloat()
                   if (f < still) {
                       stillSize = s
                   }
               }
           }
      }
      return stillSize ?: Size(100,100)
  }
}
```

The last and maybe most complex class we need is `CameraSession`. It is a state machine that handles the various camera states including autofocus and auto-exposure, and it serves two data drains: the preview texture and the captured still image storage. Before I explain a couple of constructs used here, I present the listing:

```
/**
 * A camera session class.
 */
class CameraSession(val handler: Handler,
       val cameraOpenCloseLock:Semaphore,
       val cameraCharacteristics:CameraCharacteristics?,
       val textureView: TextureView,
       val imageReader: ImageReader,
       val cameraDevice: CameraDevice,
       val previewDim: PreviewDimension,
       val rotation:Int,
       val activeArraySize: Rect,
       val zoom: Double = 1.0) {
  companion object {
      val LOG_KEY = "Session"
```

```
    enum class State {
        STATE_PREVIEW,
            // Showing camera preview.
        STATE_WAITING_LOCK,
            // Waiting for the focus to be locked.
        STATE_WAITING_PRECAPTURE,
            // Waiting for the exposure to be
            // precapture state.
        STATE_WAITING_NON_PRECAPTURE,
            // Waiting for the exposure state to
            // be something other than precapture
        STATE_PICTURE_TAKEN
            // Picture was taken.
    }
}

var mState:State = State.STATE_PREVIEW
```

The inner class `MyCaptureCallback` is responsible for handling both cases, the preview and the still image capture. For the preview, however, state transitions are limited to `on` and `off`.

```
inner class MyCaptureCallback :
      CameraCaptureSession.CaptureCallback() {
    private fun process(result: CaptureResult) {
      if(captSess == null)
          return
      when (mState) {
        State.STATE_PREVIEW -> {
            // We have nothing to do when the
            // camera preview is working normally.
        }
        State.STATE_WAITING_LOCK -> {
            val afState = result.get(
                  CaptureResult.CONTROL_AF_STATE)
            if (CaptureResult.
                  CONTROL_AF_STATE_FOCUSED_LOCKED
                  == afState
                || CaptureResult.
                   CONTROL_AF_STATE_NOT_FOCUSED_LOCKED
                   == afState
                   || CaptureResult.
                    CONTROL_AF_STATE_PASSIVE_FOCUSED
                  == afState) {
                if(cameraHasAutoExposure) {
                    mState =
                      State.STATE_WAITING_PRECAPTURE
                    runPrecaptureSequence()
                } else {
```

```
                    mState =
                      State.STATE_PICTURE_TAKEN
                    captureStillPicture()
                }
            }
        }
        State.STATE_WAITING_PRECAPTURE -> {
            val aeState = result.get(
                  CaptureResult.CONTROL_AE_STATE)
            if (aeState == null ||
                  aeState == CaptureResult.
                    CONTROL_AE_STATE_PRECAPTURE
                  ||
                  aeState == CaptureRequest.
                    CONTROL_AE_STATE_FLASH_REQUIRED) {
                mState =
                    State.STATE_WAITING_NON_PRECAPTURE
            }
        }
        State.STATE_WAITING_NON_PRECAPTURE -> {
            val aeState = result.get(
                  CaptureResult.CONTROL_AE_STATE)
            if (aeState == null ||
                  aeState != CaptureResult.
                      CONTROL_AE_STATE_PRECAPTURE) {
                mState = State.STATE_PICTURE_TAKEN
                captureStillPicture()
            }
        }
        else -> {}
    }
  }

  override
  fun onCaptureProgressed(
        session: CameraCaptureSession,
        request: CaptureRequest,
        partialResult: CaptureResult) {
      //...
  }

  override
  fun onCaptureCompleted(
        session: CameraCaptureSession,
        request: CaptureRequest,
        result: TotalCaptureResult) {
      process(result)
  }
}
```

```
var captSess: CameraCaptureSession? = null
var cameraHasAutoFocus = false
var cameraHasAutoExposure = false
val captureCallback = MyCaptureCallback()

private val stillImageTakenConsumers =
      mutableListOf<() -> Unit>()
fun addStillImageTakenConsumer(l: () -> Unit) {
    stillImageTakenConsumers.add(l)
}
```

An autofocus action is limited to camera devices supporting it. This is checked at the beginning of `createCameraSession()`. Likewise, an auto-exposure action is limited to appropriate devices.

```
/**
 * Creates a new [CameraCaptureSession] for camera
 * preview and taking pictures.
 */
fun createCameraSession() {
    //Log.e(LOG_KEY,"Starting preview session")

    cameraHasAutoFocus = cameraCharacteristics?.
          get(CameraCharacteristics.
          CONTROL_AF_AVAILABLE_MODES)?.let {
        it.any{ it ==
              CameraMetadata.CONTROL_AF_MODE_AUTO }
    } ?: false

    cameraHasAutoExposure = cameraCharacteristics?.

        get(CameraCharacteristics.
        CONTROL_AE_AVAILABLE_MODES)?.let {
      it.any{ it == CameraMetadata.
              CONTROL_AE_MODE_ON ||
          it == CameraMetadata.
              CONTROL_AE_MODE_ON_ALWAYS_FLASH ||
          it == CameraMetadata.
              CONTROL_AE_MODE_ON_AUTO_FLASH ||
          it == CameraMetadata.
              CONTROL_AE_MODE_ON_AUTO_FLASH_REDEYE }
    } ?: false

    try {
        val texture = textureView.getSurfaceTexture()
        // We configure the size of default buffer
        // to be the size of camera preview we want.
        texture.setDefaultBufferSize(
              previewDim.previewSize!!.width,
              previewDim!!.previewSize!!.height)
        // This is the output Surface we need to start
        // preview.
```

```
val previewSurface = Surface(texture)
val takePictureSurface = imageReader.surface
```

There are two camera output consumers: the texture for the preview and an image reader for the still image capture. Both are constructor parameters, and both are used for creating the session object; see `cameraDevice.createCaptureSession(...)`.

```
// Here, we create a CameraCaptureSession for
// both camera preview and taking a picture
cameraDevice.
createCaptureSession(Arrays.asList(
      previewSurface, takePictureSurface),
    object : CameraCaptureSession.
             StateCallback() {
        override
        fun onConfigured(cameraCaptureSession:
              CameraCaptureSession) {
            // When the session is ready, we
            // start displaying the preview.
            captSess = cameraCaptureSession
            try {

            val captReq =
              buildPreviewCaptureRequest()
            captSess?.
              setRepeatingRequest(captReq,
                    captureCallback,
                    handler)
        } catch (e: Exception) {
            Log.e(LOG_KEY,
            "Cannot access camera "+
            "in onConfigured()", e)
        }
    }
    override fun onConfigureFailed(
            cameraCaptureSession:
            CameraCaptureSession) {
        Log.e(LOG_KEY,
            "Camera Configuration Failed")
    }
    override fun onActive(
          sess: CameraCaptureSession) {
    }
    override fun onCaptureQueueEmpty(
          sess: CameraCaptureSession) {
    }
    override fun onClosed(
          sess: CameraCaptureSession) {
    }
    override fun onReady(
          sess: CameraCaptureSession) {
    }
```

```
                override fun onSurfacePrepared(
                      sess: CameraCaptureSession, surface: Surface) {
                }
            }, handler
        )
    } catch (e: Exception) {
        Log.e(LOG_KEY, "Camera access failed", e)
    }
}

/**
 * Initiate a still image capture.
 */
fun takePicture() {
    lockFocusOrTakePicture()
}

fun close() {
    try {
        cameraOpenCloseLock.acquire()
        captSess?.run {
            stopRepeating()
            abortCaptures()
            close()
            captSess = null
        }
        cameraDevice.run {
            close()
        }
    } catch (e: InterruptedException) {
        Log.e(LOG_KEY,
              "Interrupted while trying to lock " +
              "camera closing.", e)
    } catch (e: CameraAccessException) {
        Log.e(LOG_KEY, "Camera access exception " +
              "while closing.", e)
    } finally {
        cameraOpenCloseLock.release()
    }
}
```

The following are the private methods. The various `build*CaptureRequest()` methods show how to prepare a request, which then get sent to the camera hardware.

```
////////////////////////////////////////////////////
////////////////////////////////////////////////////

private fun buildPreviewCaptureRequest():
      CaptureRequest {
    val texture = textureView.getSurfaceTexture()
    val surface = Surface(texture)
```

```
    // We set up a CaptureRequest.Builder with the
    // preview output Surface.

    val reqBuilder = cameraDevice.
          createCaptureRequest(
          CameraDevice.TEMPLATE_PREVIEW)
    reqBuilder.addTarget(surface)

    // Zooming
    val cropRect = calcCropRect()
    reqBuilder.set(
          CaptureRequest.SCALER_CROP_REGION,
          cropRect)

    // Flash off
    reqBuilder.set(CaptureRequest.FLASH_MODE,
          CameraMetadata.FLASH_MODE_OFF)

    // Continuous autofocus
    reqBuilder.set(CaptureRequest.CONTROL_AF_MODE,
          CaptureRequest.
          CONTROL_AF_MODE_CONTINUOUS_PICTURE)
    return reqBuilder.build()
}

private fun buildTakePictureCaptureRequest() :
      CaptureRequest {
    // This is the CaptureRequest.Builder that we use
    // to take a picture.
    val captureBuilder =
          cameraDevice.createCaptureRequest(
          CameraDevice.TEMPLATE_STILL_CAPTURE)
    captureBuilder.addTarget(imageReader.getSurface())

    // Autofocus mode
    captureBuilder.set(CaptureRequest.CONTROL_AF_MODE,
            CaptureRequest.
            CONTROL_AF_MODE_CONTINUOUS_PICTURE)

    // Flash auto
    captureBuilder.set(CaptureRequest.CONTROL_AE_MODE,
            CaptureRequest.
            CONTROL_AE_MODE_ON_AUTO_FLASH)
    // captureBuilder.set(CaptureRequest.FLASH_MODE,
    // CameraMetadata.FLASH_MODE_OFF)

        // Zoom
        val cropRect = calcCropRect()
        captureBuilder.set(CaptureRequest.
              SCALER_CROP_REGION, cropRect)
```

```
        // Orientation
        captureBuilder.set(CaptureRequest.
              JPEG_ORIENTATION,
              previewDim.getOrientation(rotation))
        return captureBuilder.build()
    }

    private fun buildPreCaptureRequest() :
          CaptureRequest {
        val surface = imageReader.surface
        val reqBuilder =
              cameraDevice.createCaptureRequest(
              CameraDevice.TEMPLATE_STILL_CAPTURE)
        reqBuilder.addTarget(surface)
        reqBuilder.set(CaptureRequest.
              CONTROL_AE_PRECAPTURE_TRIGGER,
              CaptureRequest. CONTROL_AE_PRECAPTURE_TRIGGER_START)
        return reqBuilder.build()
    }

    private fun buildLockFocusRequest() :
          CaptureRequest {
        val surface = imageReader.surface
        val reqBuilder =
              cameraDevice.createCaptureRequest(
              CameraDevice.TEMPLATE_STILL_CAPTURE)
        reqBuilder.addTarget(surface)
        reqBuilder.set(CaptureRequest.
              CONTROL_AF_TRIGGER,
              CameraMetadata.CONTROL_AF_TRIGGER_START)
        return reqBuilder.build()
    }

    private fun buildCancelTriggerRequest() :
          CaptureRequest {
        val texture = textureView.getSurfaceTexture()
        val surface = Surface(texture)

        val reqBuilder =
              cameraDevice.createCaptureRequest(
              CameraDevice.TEMPLATE_PREVIEW)
        reqBuilder.addTarget(surface)
        reqBuilder.set(CaptureRequest.CONTROL_AF_TRIGGER,
              CameraMetadata.CONTROL_AF_TRIGGER_CANCEL)
        return reqBuilder.build()
    }
```

Capturing a still picture gets handled by the method `captureStillPicture()`. Note that, like for many of the other camera-related functionalities, appropriate tasks get sent to the background, and callbacks handle the background processing results.

```
private fun captureStillPicture() {
    val captureRequest =
          buildTakePictureCaptureRequest()
    if (captSess != null) {
      try {
        val captureCallback = object :
              CameraCaptureSession.CaptureCallback() {
            override fun onCaptureCompleted(
                  session: CameraCaptureSession,
                  request: CaptureRequest,
                  result: TotalCaptureResult) {
                //Util.showToast(activity,
                //"Acquired still image")
                stillImageTakenConsumers.forEach {
                      it() }
                unlockFocusAndBackToPreview()
            }
        }
        captSess?.run {
            stopRepeating()
            capture(captureRequest,
                  captureCallback, null)
        }
      } catch (e: Exception) {
          Log.e(LOG_KEY,
                "Cannot capture picture", e)
      }
    }
}

private fun lockFocusOrTakePicture() {
    if(cameraHasAutoFocus) {
        captSess?.run {
          try {
            val captureRequest =
                  buildLockFocusRequest()
            mState = State.STATE_WAITING_LOCK
                capture(captureRequest,
                        captureCallback,
                        handler)
          } catch (e: Exception) {
              Log.e(LOG_KEY,
                    "Cannot lock focus", e)
          }
        }
```

```
    } else {
        if(cameraHasAutoExposure) {
            mState = State.STATE_WAITING_PRECAPTURE
            runPrecaptureSequence()
        } else {
            mState = State.STATE_PICTURE_TAKEN
            captureStillPicture()
        }
    }
}

/**
 * Unlock the focus. This method should be called when
 * still image capture sequence is finished.
 */
private fun unlockFocusAndBackToPreview() {
    captSess?.run {
        try {
            mState = State.STATE_PREVIEW
            val cancelAfTriggerRequest =
                  buildCancelTriggerRequest()
            val previewRequest =
                  buildPreviewCaptureRequest()
            capture(cancelAfTriggerRequest,
                    captureCallback,
                    handler)
            setRepeatingRequest(previewRequest,
                    captureCallback,
                    handler)
        } catch (e: Exception) {
            Log.e(LOG_KEY,
            "Cannot go back to preview mode", e)
        }
    }
}
```

Running the precapture sequence for capturing a still image gets performed by the method `runPrecaptureSequence()`. This method should be called when we get a response in `captureCallback` from the method `lockFocusThenTakePicture()`.

```
  /**
   * Run the precapture sequence for capturing a still
   * image.
   */
  private fun runPrecaptureSequence() {
      try {
        captSess?.run {
           val captureRequest = buildPreCaptureRequest()
           mState = State.STATE_WAITING_PRECAPTURE
           capture(captureRequest, captureCallback,
                   handler)
         }
```

```
        } catch (e: Exception) {
            Log.e(LOG_KEY, "Cannot access camera", e)
        }
    }

    private fun calcCropRect(): Rect {
        with(activeArraySize) {
            val cropW = width() / zoom
            val cropH = height() / zoom
            val top = centerY() - (cropH / 2f).toInt()
            val left = centerX() - (cropW / 2f).toInt()
            val right = centerX() + (cropW / 2f).toInt()
            val bottom = centerY() + (cropH / 2f).toInt()
            return Rect(left, top, right, bottom)
        }
    }
}
```

Here are a couple of notes on the `CameraSession` class:

- The emulators don't exhibit autofocus capabilities. The code takes care of that.
- The term *precapture* is just another name for *auto-exposure*.
- Using the flash is a `todo` in this class. To enable flashing, see the places where the flash gets mentioned in the code.
- By virtue of a listener chain starting in `CameraSession`, the still image capture data eventually arrives in the `dataArrived(...)` method of `MainActivity`. It is there where you can start writing further processing algorithms such as saving, sending, converting, reading, and so on.

Android and NFC

NFC adapters, provided the Android device has one, allow for short-range wireless communication with other NFC-capable devices or NFC tags. We talked about NFC in Chapter 12.

Android and Bluetooth

Most if not all modern Android devices have Bluetooth built in. Via Bluetooth they can wirelessly communicate with other Bluetooth devices. For details, please see Chapter 12.

Android Sensors

Android devices provide various bits of information about their environment to apps, listed here:

- Orientation as determined by a compass or gyroscope
- Motion as given by acceleration forces
- Gravitational forces
- Air temperature, pressure, humidity
- Illumination
- Proximity, for example to find out the distance to the user's ear

The exact geospatial position of a device is not detected by a sensor. For the detection of positional coordinates using GPS, instead see Chapter 8.

Retrieving Sensor Capabilities

Beginning with Android 4.0 (API level 14), Android devices are supposed to provide all sensor types as defined by the various `android.hardware.Sensor.TYPE_*` constants. To see a list of all sensors including various information about them, use the following code snippet:

```
val sensorManager = getSystemService(
      Context.SENSOR_SERVICE) as SensorManager
val deviceSensors =
      sensorManager.getSensorList(Sensor.TYPE_ALL)
deviceSensors.forEach { sensor ->
  Log.e("LOG", "+++" + sensor.toString())
}
```

To fetch a certain sensor instead, use the following:

```
val magneticFieldSensor = sensorManager.getDefaultSensor(
      Sensor.TYPE_MAGNETIC_FIELD)
```

Once you have a `Sensor` object, you can obtain various information about it. Please see the API documentation of `android.hardware.Sensor` for details. To find out sensor values, see the following section.

Listening to Sensor Events

Android allows for the following two sensor event listeners:

- Changes in a sensor's accuracy
- Changes in a sensor's value

To register for a listener, fetch the sensor manager and the sensor, as described in the preceding section, and then use something like the following inside the activity:

```
val sensorManager = getSystemService(
      Context.SENSOR_SERVICE) as SensorManager
val magneticFieldSensor = sensorManager.getDefaultSensor(
      Sensor.TYPE_MAGNETIC_FIELD)
sensorManager.registerListener(this,
      magneticFieldSensor,
      SensorManager.SENSOR_DELAY_NORMAL)
```

For the temporal resolution, you can also use one of the other delay specifications: `SensorManager.SENSOR_DELAY_*`.

The activity must then overwrite `android.hardware.SensorEventListener` and implement it.

```
class MainActivity : AppCompatActivity(),
      SensorEventListener {
  private lateinit var sensorManager:SensorManager
  private lateinit var magneticFieldSensor:Sensor

  override fun onCreate(savedInstanceState: Bundle?) {
      ...
      sensorManager =
            getSystemService(Context.SENSOR_SERVICE)
            as SensorManager
      magneticFieldSensor =
            sensorManager.getDefaultSensor(
            Sensor.TYPE_MAGNETIC_FIELD)
  }

  override
  fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {
      // Do something here if sensor accuracy changes.
  }

  override
  fun onSensorChanged(event: SensorEvent) {
      Log.e("LOG", Arrays.toString(event.values))
      // Do something with this sensor value.
  }

  override
  fun onResume() {
      super.onResume()
      sensorManager.registerListener(this,
            magneticFieldSensor,
      SensorManager.SENSOR_DELAY_NORMAL)
  }
```

```
  override
  fun onPause() {
      super.onPause()
      sensorManager.unregisterListener(this)
  }
}
```

As shown in this example, it is important to unregister sensor event listeners when no longer needed since sensors may substantially drain battery power.

> **Note** Other than the name suggests, `onSensorChanged` events might be fired even when there is not really a sensor value change.

All possible sensor values you get from `SensorEvent.values` inside `onSensorChanged()` are listed in Table 13-3.

***Table 13-3.** Sensor Event Values*

Type	Values
`TYPE_ACCELEROMETER`	Vector3: Acceleration along the x-y-z axes in m/s^2. Includes gravity.
`TYPE_AMBIENT_TEMPERATURE`	Scalar: The ambient air temperature in *°C*.
`TYPE_GRAVITY`	Vector3: Gravitational force along the x-y-z axes in m/s^2.
`TYPE_GYROSCOPE`	Vector3: Rate of rotation around each of the x-y-z axes, in *rad/s*.
`TYPE_LIGHT`	Scalar: Illuminance in *lx*.
`TYPE_LINEAR_ACCELERATION`	Vector3: Acceleration along the x-y-z axes in m/s^2. Without gravity.
`TYPE_MAGNETIC_FIELD`	Vector3: Strength of the geomagnetic field in *μT*.
`TYPE_ORIENTATION`	Vector3: Azimutz, pitch, roll in degrees.
`TYPE_PRESSURE`	Scalar: Ambient air pressure in *hP a*.
`TYPE_PROXIMITY`	Scalar: Distance from object in *cm*.
`TYPE_RELATIVE_HUMIDITY`	Scalar: Ambient relative humidity in %.
`TYPE_ROTATION_VECTOR`	Vector4: Rotation vector as a quaternion.
`TYPE_SIGNIFICAT_MOTION`	The event gets fired each time a significant motion is detected. To catch this event, you must register via `SensorManager.requestTriggerSensor(...)`.
`TYPE_STEP_COUNTER`	Scalar: Accumulated step count since reboot and while the sensor is activated.
`TYPE_STEP_DETECTOR`	The event gets fired each time a step is detected.
`TYPE_TEMPERATURE`	Deprecated. Scalar: The device's temperature in *°C*.

Some sensors have an uncalibrated version, which means they show changes more accurately but less accurately relate to a fixed point:

- TYPE_ACCELEROMETER_UNCALIBRATED
- TYPE_GYROSCOPE_UNCALIBRATED
- TYPE_MAGNETIC_FIELD_UNCALIBRATED.

Instead of the TYPE_ROTATION_VECTOR sensor, you can also use one of the following:

- TYPE_GAME_ROTATION_VECTOR
- TYPE_GEOMAGNETIC_ROTATION_VECTOR

The first one does not use a gyroscope and is more accurate for detecting changes, but not so accurate to find out where north is. The second one uses the magnetic field instead of a gyroscope; it is less accurate but also needs less battery power.

Interacting with Phone Calls

Android allows for a couple of ways to interact with incoming or outgoing phone calls and the dialing process. The following are the most prominent use cases your app might implement for telephony:

- Monitor state changes of the telephone, like being informed of incoming and outgoing calls
- Initiate a dialing process to start outgoing calls
- Provide its own UI for managing a call

You can find telephony relevant classes and interfaces inside the packages android.telecom and android.telephony and their subpackages.

Monitoring Phone State Changes

To monitor phone state changes, add the following permissions to AndroidManifest.xml:

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

The READ_PHONE_STATE permission allows you to detect the status of ongoing calls. The PROCESS_OUTGOING_CALLS permission lets your app see the number of outgoing calls or even use a different number or cancel calls.

To learn how to acquire permissions from within your app, please see Chapter 7.

To listen to phone-related events, you then add a broadcast receiver inside `AndroidManifest.xml`.

```
<application>
  ...
  <receiver android:name=".CallMonitor">
    <intent-filter>
      <action android:name=
            "android.intent.action.PHONE_STATE" />
      </intent-filter>
      <intent-filter>
      <action android:name=
            "android.intent.action.NEW_OUTGOING_CALL" />
      </intent-filter>
  </receiver>
</application>
```

You implement it, for example, as follows:

```
package ...
import android.telephony.TelephonyManager as TM
import ...

class CallMonitor : BroadcastReceiver() {
    companion object {
        private var lastState = TM.CALL_STATE_IDLE
        private var callStartTime: Date? = null
        private var isIncoming: Boolean = false
        private var savedNumber: String? = null
    }
```

The `onReceive()` callback handles incoming broadcasts, this time an incoming or outgoing call.

```
override
fun onReceive(context: Context, intent: Intent) {
    if (intent.action ==
          Intent.ACTION_NEW_OUTGOING_CALL) {
        savedNumber = intent.extras!!.
              getString(Intent.EXTRA_PHONE_NUMBER)
    } else {
        val stateStr = intent.extras!!.
              getString(TM.EXTRA_STATE)
        val number = intent.extras!!.
              getString(TM.EXTRA_INCOMING_NUMBER)
        val state = when(stateStr) {
            TM.EXTRA_STATE_IDLE ->
                TM.CALL_STATE_IDLE
            TM.EXTRA_STATE_OFFHOOK ->
                TM.CALL_STATE_OFFHOOK
            TM.EXTRA_STATE_RINGING ->
                TM.CALL_STATE_RINGING
            else -> 0
        }
```

```
        callStateChanged(context, state, number)
    }
}

protected fun onIncomingCallReceived(
      ctx: Context, number: String?, start: Date){
    Log.e("LOG",
          "IncomingCallReceived ${number} ${start}")
}

protected fun onIncomingCallAnswered(
      ctx: Context, number: String?, start: Date) {
    Log.e("LOG",
          "IncomingCallAnswered ${number} ${start}")
}

protected fun onIncomingCallEnded(
      ctx: Context, number: String?,
      start: Date?, end: Date) {
    Log.e("LOG",
          "IncomingCallEnded ${number} ${start}")
}

protected fun onOutgoingCallStarted(
      ctx: Context, number: String?, start: Date) {
    Log.e("LOG",
          "OutgoingCallStarted ${number} ${start}")
}

protected fun onOutgoingCallEnded(
      ctx: Context, number: String?,
      start: Date?, end: Date) {
    Log.e("LOG",
          "OutgoingCallEnded ${number} ${start}")
}

protected fun onMissedCall(
      ctx: Context, number: String?, start: Date?) {
    Log.e("LOG",
          "MissedCall ${number} ${start}")
}
```

The private method `callStateChanged()` reacts on the various state changes corresponding to phone calls.

```
/**
 * Incoming call:
 *     IDLE -> RINGING when it rings,
 *     -> OFFHOOK when it's answered,
 *     -> IDLE when its hung up
 * Outgoing call:
 *     IDLE -> OFFHOOK when it dials out,
```

```
 *     -> IDLE when hung up
 * */
private fun callStateChanged(
      context: Context, state: Int, number: String?) {
    if (lastState == state) {
        return // no change in state
    }
    when (state) {
        TM.CALL_STATE_RINGING -> {
            isIncoming = true
            callStartTime = Date()
            savedNumber = number
            onIncomingCallReceived(
                  context, number, callStartTime!!)
        }
        TM.CALL_STATE_OFFHOOK ->
            if (lastState != TM.CALL_STATE_RINGING) {
                isIncoming = false
                callStartTime = Date()
                onOutgoingCallStarted(context,
                      savedNumber, callStartTime!!)
            } else {
                isIncoming = true
                callStartTime = Date()
                onIncomingCallAnswered(context,
                      savedNumber, callStartTime!!)
            }
        TM.CALL_STATE_IDLE ->
            if (lastState == TM.CALL_STATE_RINGING) {
                //Ring but no pickup- a miss
                onMissedCall(context,
                      savedNumber, callStartTime)
                } else if (isIncoming) {
                    onIncomingCallEnded(context,
                    savedNumber, callStartTime,
                    Date())
            } else {
                onOutgoingCallEnded(context,
                      savedNumber, callStartTime,
                      Date())
            }
        }
       lastState = state
    }
}
```

Using such a listener, you can gather statistical information about phone usage, create a priority phone number list, or do other interesting things. To connect phone calls with contact information, see Chapter 8.

Initiate a Dialing Process

To initiate a dialing process from within your app, you basically have two options.

- Start a dialing process; the user sees and can change the called number.
- Start a dialing process; the user cannot change the called number.

For the first case, showing the user the number and letting them change it, you don't need any special permission. Just write the following:

```
val num = "+34111222333"
val intent = Intent(Intent.ACTION_DIAL,
      Uri.fromParts("tel", num, null))
startActivity(intent)
```

To start a dialing process with a prescribed number, you need the following as an additional permission:

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

To learn how to acquire it, see Chapter 7. The calling process then can be initiated via the following:

```
val num = "+34111222333"
val intent = Intent(Intent.ACTION_CALL,
      Uri.fromParts("tel", num, null))
startActivity(intent)
```

Create a Phone Call Custom UI

Creating your own phone-calling activity including its own UI is described in the online page "Build a calling app" in the Android documentation.

Fingerprint Authentication

Fingerprint authentication entered the Android framework with Android version 6.0 (API level 23). Before that, you had to use vendor-specific APIs. The following assumes you are targeting Android 6.0 or newer.

Using a fingerprint scanner makes sense only if your user's device has one. To check whether this is the case, use the following snippet:

```
val useFingerprint =
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
    (getSystemService(Context.FINGERPRINT_SERVICE)
     as FingerprintManager).let {
```

```
        it.isHardwareDetected &&
        it.hasEnrolledFingerprints()
    }
  } else false
```

This is deprecated in Android P. The substitute starting at Android P is trying to perform an authentication and catch appropriate error messages.

To now actually start a fingerprint authentication process, you first must decide whether you want to use the now deprecated `FingerPrintManager` class or the new `FingerprintDialog` starting at Android P.

To use the deprecated `FingerPrintManager` class for authentication, you can provide a callback and then call the `authenticate(...)` method on it.

```
val mngr = getSystemService(Context.FINGERPRINT_SERVICE)
      as FingerprintManager
val cb = object :
      FingerprintManager.AuthenticationCallback() {
  override
  fun onAuthenticationSucceeded(
      result: FingerprintManager.AuthenticationResult) {
      ...
  }
  override
  fun onAuthenticationFailed() {
      ...
  }
}
val cs = CancellationSignal()
mngr.authenticate(null, cs, 0, cb, null)
```

To instead use `FingerprintDialog`, you similarly start an authentication process, calling `authenticate()` and reacting to authentication results appropriately.

> **Note** As of April 2018, there only exists a developer preview of `FingerprintDialog`.

Chapter 14

Testing

A lot has been said about testing in information technology. There are three reasons for the attention testing has gained during the last decades.

- Testing is the interface between the developers and the users.
- Testing can be engineered to some extent.
- Testing helps increase profits.

Developers tend to have a biased view of their software. No offense is intended by saying that. It is just natural that if you spend a lot of time with some subject, you potentially lose the ability to anticipate what is going on in a new user's mind. It is therefore strongly advised to regularly step out of your developer role and ask yourself the question, "Suppose I didn't know anything about the app—if I enter this GUI workflow, does it make sense, is it easy to follow, and is it hard to make unrecoverable mistakes?" Testing helps with that. It forces the developer to take on this end-user role and ask this question.

Development is far from being an industrially engineered science. This is good news and bad news. If it had a strong engineering path, it would be easier to follow agreed-on development patterns, and other developers would much more readily understand what you are doing. On the other hand, not being that precisely engineerable also opens the realm to more creativity and allows for development to become an art. Good developers know that they are constantly swaying between those limits. Testing nowadays tends to prioritize engineerability. This stems from the fact that you can precisely say what software is supposed to do, totally ignorant of a single line of code. So, parts of the tests just don't care how things were accomplished on the coding level, taking away the plethora of possibilities of how development demands were satisfied. This is not true for low-level unit tests, but even for those you can see a strong overlap of software artifact contracts and testing methodologies. So, testing the grade of engineerability is somewhat higher compared to mere developing. However, because testing is just one aspect of the development process, it is still possible to have an interesting job as a developer and live in both worlds. You can be an artist during developing code and an engineer while writing tests.

P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_14

On the other side of the development chain, depending on your intention, you might want to have end users spend some money for your app. Testing obviously helps to avoid frustration because of bugs you didn't anticipate, allowing for the public to more readily buy your app.

A lot has been said about testing for Android, and you can find good information and introductory or advanced-level videos in the official Android documentation. The rest of this chapter should be seen as advice and a collection of empirical know-how on the matter of testing. I do not intend to give an introduction to testing, covering each and every aspect of it, but I hope I can give you a starting point for your own, deeper research.

Unit Tests

Unit tests aim at the class level and test low-level functional aspects of your app. By "functional" I mean unit tests usually check deterministic relations between input and output of a method call, maybe but not necessarily including state variables of class instances in a deterministic, straightforward manner.

Standard Unit Tests

In an Android environment, standard unit tests run without dependency on the device hardware or any Android framework classes and are thus executed on the development machine.

They are typically useful for libraries and not so much GUI-related functionalities, which is why the applicability of this kind of unit test is somewhat limited for most Android apps.

However, if your app's classes contain method calls and you can anticipate the call result given various sets of inputs, using standard unit tests makes sense. It is easy to add unit tests to your app. In fact, if you start a new project using Android Studio, unit testing is all set up for you, and you even get a sample test class, as shown in Figure 14-1.

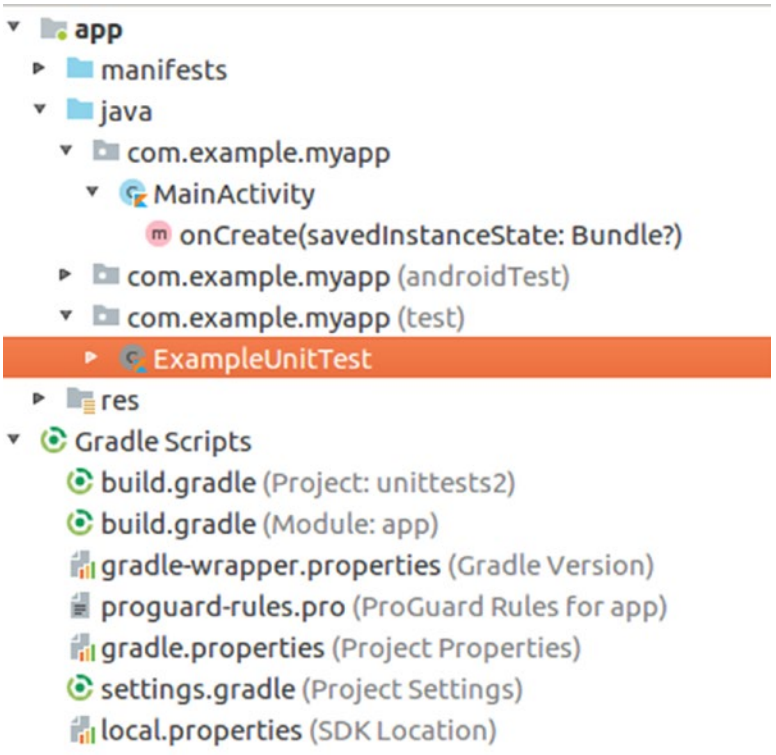

Figure 14-1. *Initial unit test setup*

So, you immediately can start writing unit tests using that test class as an example; just add more test classes to the `test` section of the source code.

> **Note** While not technically necessary, a common convention is to use the same names for the test classes as the classes under test, with `Test` appended. So, the test class for `com.example.myapp.TheClass` should be called `com.example.myapp.TheClassTest`.

To run the unit tests inside Android Studio, right-click the `test` section and select Run Tests in or Debug Tests in.

Unit Tests with Stubbed Android Framework

By default, the Gradle plugin used for executing unit tests contains a stubbed version of the Android framework, throwing an exception whenever an Android class gets called.

You can change this behavior by adding the following to the app's `build.gradle` file:

```
android {
  ...
  testOptions {
    unitTests.returnDefaultValues = true
  }
}
```

Any call of an Android class's method then does nothing and returns `null` on demand.

Unit Tests with Simulated Android Framework

If you need to access Android classes from inside your unit tests and expect them to do real things, using the community-supported Robolectric frameworks as a unit test implementation is a valid option. With Robolectric you can simulate clicking buttons, reading and writing text, and lots of other GUI-related activities. Still, all that runs on your development machine, which considerably speeds up testing.

To allow your project to use Robolectric, add the following to your app's `build.gradle` file:

```
android {
  testOptions {
      unitTests {
          includeAndroidResources = true
      }
  }
}

dependencies {
  ...
  //testImplementation 'junit:junit:4.12'
  testImplementation "org.robolectric:robolectric:3.8"
}
```

As an example, a test class that simulates the click on a `Button` and then checks whether the click action has updated a `TextView` looks like this:

```
package com.example.robolectric

import org.junit.runner.RunWith
import org.robolectric.RobolectricTestRunner
import org.robolectric.shadows.ShadowApplication
import android.content.Intent
import android.widget.Button
import android.widget.TextView
import org.junit.Test
import org.robolectric.Robolectric
import org.junit.Assert.*
```

```
@RunWith(RobolectricTestRunner::class)
class MainActivityTest {
  @Test
  fun clickingGo_shouldWriteToTextView() {
      val activity = Robolectric.setupActivity(
            MainActivity::class.java!!)
      activity.findViewById<Button>(R.id.go).
            performClick()
      assertEquals("Clicked",
            activity.findViewById<TextView>(
            R.id.tv).text)
  }
}
```

You start that test like any normal unit test by right-clicking the `test` section and selecting Run Tests in or Debug Tests in.

For more test options and details, please see the home page of Robolectric at www.robolectric.org.

Unit Tests with Mocking

Mocking means you let the test hook into the call of the Android OS functions and simulate their execution by mimicking their functioning.

If you want to include mocking in unit tests, the Android developer documentation suggests that you use the Mockito test library. I suggest going one step further and using PowerMock instead, which sits on top of Mockito but adds more power to it like mocking of static or final classes.

To enable PowerMock, add the following to your app's `build.gradle` file (remove the line break after `powermock:`):

```
android {
  ...
  testOptions {
      unitTests.returnDefaultValues = true
  }
}

dependencies {
  ...
  testImplementation ('org.powermock:
        powermock-mockito-release-full:1.6.1') {
      exclude module: 'hamcrest-core'
      exclude module: 'objenesis'
  }
  testImplementation 'org.reflections:reflections:0.9.11'
 }
}
```

Do not remove or comment out the `testImplementation 'junit:junit:4.12'` line inside the `dependencies` section, because it still is needed. The `unitTests.returnDefaultValues = true` entry takes care of the stub Android implementation for unit tests not to throw exceptions, just in case. The `reflections` package is for scanning through packages to search for test classes.

As a nontrivial example, I present an activity that writes an entry to a database. We are going to mock out the actual database implementation but still want to make sure necessary tables get created and the `insert` statement gets executed. The activity looks like the following:

```
class MainActivity : AppCompatActivity() {

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

  fun save(v: View) {
      saveInDb(et.text.toString())
  }

  fun count(v: View) {
      val db = openOrCreateDatabase("MyDb",
            MODE_PRIVATE, null)
      with(db) {
          val resultSet = rawQuery(
                "Select * from MyItems", null)
          val cnt = resultSet.count
          Toast.makeText(this@MainActivity,
                "Count: ${cnt}", Toast.LENGTH_LONG).
                show()
      }
      db.close()
}

private fun saveInDb(item:String) {
    val tm = System.currentTimeMillis() / 1000
    val db = openOrCreateDatabase("MyDb",
          MODE_PRIVATE, null)
    with(db) {
        execSQL("CREATE TABLE IF NOT EXISTS " +
              "MyItems(Item VARCHAR,timestamp INT);")
        execSQL("INSERT INTO MyItems VALUES(?,?);",
              arrayOf(item, tm))
    }
    db.close()
  }
}
```

The corresponding layout file reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  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="com.example.powermock.MainActivity"
  android:orientation="vertical">

  <EditText
      android:id="@+id/et"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text=""/>
  <Button
      android:id="@+id/btnSave"
      android:text="Save"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:onClick="save"/>
  <Button
      android:id="@+id/btnCount"
      android:text="Count"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:onClick="count"/>
</LinearLayout>
```

It contains an `EditText` view element with ID `et` and two buttons that call the methods `save()` and `count()` of the activity.

For the test itself, create a class `MainActivityTest` inside the `test` section of the sources. Let it read as follows:

```
import android.database.sqlite.SQLiteDatabase
import org.junit.Test
import org.junit.runner.RunWith
import org.mockito.ArgumentMatcher
import org.powermock.core.classloader.annotations.
      PrepareForTest
import org.powermock.modules.junit4.PowerMockRunner
import org.mockito.BDDMockito.*
import org.mockito.Matchers
import org.powermock.reflect.Whitebox

@RunWith(PowerMockRunner::class)
@PrepareForTest(MainActivity::class)
class MainActivityTest {
```

```
@Test
fun table_created() {
    val activity = MainActivity()
    val activitySpy = spy(activity)
    val db = mock(SQLiteDatabase::class.java)

    // given
    given(activitySpy.openOrCreateDatabase(
          anyString(), anyInt(), any())).willReturn(db)

    // when
    Whitebox.invokeMethod<Unit>(
          activitySpy,"saveInDb","hello")

    // then
    verify(db).execSQL(Matchers.argThat(
          object : ArgumentMatcher<String>() {
        override
        fun matches(arg:Any):Boolean {
            return arg.toString().matches(
            Regex("(?i)create table.*\\bMyItems\\b.*"))
        }
    }))

}

    @Test
    fun item_inserted() {
        val activity = MainActivity()
        val activitySpy = spy(activity)
        val db = mock(SQLiteDatabase::class.java)

        // given
        given(activitySpy.openOrCreateDatabase(
              anyString(), anyInt(), any())).willReturn(db)

        // when
        Whitebox.invokeMethod<Unit>(
              activitySpy,"saveInDb","hello")

        // then
        verify(db).execSQL(Matchers.argThat(
              object : ArgumentMatcher<String>() {
            override
            fun matches(arg:Any):Boolean {
                return arg.toString().matches(
                Regex("(?i)insert into MyItems\\b.*"))
            }
        }), Matchers.argThat(
              object : ArgumentMatcher<Array<Any>>() {
            override
            fun matches(arg:Any):Boolean {
```

```
                val arr = arg as Array<Any>
                return arr[0] == "hello" &&
                      arr[1] is Number
            }
        }))
    }
}
```

`@RunWith(PowerMockRunner::class)` will make sure PowerMock gets used as a unit test runner, and `@PrepareForTest(MainActivity::class)` prepares the `MainActivity` class, so it can be mocked even though it is marked `final` (something that Kotlin does by default).

The first function, `table_created()`, is supposed to make sure the table gets created if necessary. It acts as follows:

- We instantiate `MainActivity`, which is possible since the instantiation does not call Android framework classes.
- We wrap the `MainActivity` instance into a *spy*. This allows us to hook into method calls to mock out the actual implementation.
- We create a mock of `SQLiteDatabase` so we can hook into database operations without actually using a real database.
- The following `//given`, `//when`, and `//then` sections follow the BDD style of development.
- Inside the `//given` section, we mock out the `openOrCreateDatabase()` call of the activity and instead let it return our mock database.
- Inside `//when` we call the `private` method `saveInDb()` of the activity class. Calling private methods is frowned upon in test development, but here we have no other chance because we can't use the `save()` method and let it access the `EditText` view without more complex work. Because of all that mocking preparation, this call reaches the real activity class but will use the mocked database instead of the real one.
- In the `//then` section, we can check whether the call of `saveInDb()` invokes the appropriate database operation to create the necessary table. For this aim we use an `ArgMatcher`, which allows us to check for appropriate method call arguments.

The test function `item_inserted()` does almost the same but instead checks whether an appropriate `insert` statement gets fired to the database.

Using PowerMock as a unit test runner with Kotlin inside Android Studio has a drawback: normally you can use the context menu of the *package* to run all unit tests inside, but for some reason that does not work for PowerMock and Kotlin. As a workaround, I present a single test class as a suite that calls all the test classes it can find in the package. That is why we added `testImplementation 'org.reflections:reflections:0.9.11'` in the Gradle build file.

```
@RunWith(TestAll.TestAllRunner::class)
class TestAll {
  class TestAllRunner(klass: Class<*>?,
        runnersO: List<Runner>) :
        ParentRunner<Runner>(klass) {
      private val runners: List<Runner>

      constructor(clazz: Class<*>) : t
            his(clazz, listOf<Runner>()) {
      }

      init {
          val classLoadersList = arrayOf(
                  ClasspathHelper.contextClassLoader(),
                  ClasspathHelper.staticClassLoader())

          val reflections = Reflections(
                ConfigurationBuilder()
                  .setScanners(SubTypesScanner(false),
                        TypeAnnotationsScanner())
                  .setUrls(ClasspathHelper.
                        forClassLoader(
                        *classLoadersList))
                  .filterInputsBy(FilterBuilder().
                        include(FilterBuilder.
                        prefix(
                        javaClass.`package`.name))))

          runners = reflections.getTypesAnnotatedWith(
                RunWith::class.java).filter {
              clazz ->
              clazz.getAnnotation(RunWith::class.java).
                    value.toString().
                    contains(".PowerMockRunner")
          }.map { PowerMockRunner(it) }
      }

      override fun getChildren(): List<Runner> = runners

      override fun describeChild(child: Runner):
            Description = child.description

      override fun runChild(runner: Runner,
            notifier: RunNotifier) {
          runner.run(notifier)
      }
    }
}
```

This class provides its own test runner implementation, which uses the `reflections` library inside the `init ...` block to scan through the package for test classes. You can now run the test on this `TestAll` class, and it will in turn run all the test classes it can find in the package.

Integration Tests

Integration tests sit between unit tests that do fine-grained testing work on the development machine and fully fledged user interface tests running on real or virtual devices. Integration tests run on a device, too, but they do not test the app as a whole but instead test selected components in an isolated execution environment.

Integration tests happen inside the `androidTest` section of the source code. You also need to add a couple of packages to the app's `build.gradle` file, as shown here (with the line breaks after `androidTestImplementation` removed):

```
dependencies {
  ...
  androidTestImplementation
      'com.android.support:support-annotations:27.1.1'
  androidTestImplementation
      'com.android.support.test:runner:1.0.2'
  androidTestImplementation
      'com.android.support.test:rules:1.0.2'
}
```

Testing Services

To test a service with a binding, write something like this:

```
@RunWith(AndroidJUnit4::class)
class ServiceTest {

  // A @Rule wraps around the test invocation - here we
  // use the 'ServiceTestRule' which makes sure the
  // service gets started and stopped correctly.
  @Rule @JvmField
  val mServiceRule = ServiceTestRule()

  @Test
  fun testWithBoundService() {
      val serviceIntent = Intent(
          InstrumentationRegistry.getTargetContext(),
          MyService::class.java
      ).apply {
          // If needed, data can be passed to the
          // service via the Intent.
          putExtra("IN_VAL", 42L)
      }

      // Bind the service and grab a reference to the
      // binder.
      val binder: IBinder = mServiceRule.
            bindService(serviceIntent)
```

```
      // Get the reference to the service
      val service: MyService =
            (binder as MyService.MyBinder).getService()

      // Verify that the service is working correctly.
      assertThat(service.add(11,27), `is`(38))
  }
}
```

This tests a simple service called `MyService` with an `add(Int, Int)` service method.

```
class MyService : Service() {
  class MyBinder(val servc:MyService) : Binder() {
      fun getService():MyService {
          return servc
      }
  }
  private val binder: IBinder = MyBinder(this)

  override fun onBind(intent: Intent): IBinder = binder

  fun add(a:Int, b:Int) = a + b
}
```

To run the integration test, right-click the `androidTest` section of the sources and choose Run Tests in. This will create and upload an APK file, by virtue of `InstrumentationRegistry.getTargetContext()` creating an integration test context, and then run the test on the device.

Testing Intent Services

Other than the official documentation claims, services based on the `IntentService` class can be subject to integration tests as well. You just cannot use `@Rule ServiceTestRule` for handling the service lifecycle because intent services have their own ideas of when to start and stop. But you can handle the lifecycle yourself. As an example, I present a test for a simple intent service working for ten seconds and continuously sending back data through a `ResultReceiver`.

The service itself reads as follows:

```
class MyIntentService() :
    IntentService("MyIntentService") {
  class MyResultReceiver(val cb: (Double) -> Unit) :
        ResultReceiver(null) {
      companion object {
          val RESULT_CODE = 42
          val INTENT_KEY = "my.result.receiver"
          val DATA_KEY = "data.key"
      }
      override
      fun onReceiveResult(resultCode: Int,
            resultData: Bundle?) {
```

```
            super.onReceiveResult(resultCode, resultData)
            val d = resultData?.get(DATA_KEY) as Double
            cb(d)
        }
    }
    var status = 0.0
    override fun onHandleIntent(intent: Intent) {
        val myReceiver = intent.
                getParcelableExtra<ResultReceiver>(
                        MyResultReceiver.INTENT_KEY)
        for (i in 0..100) {
            Thread.sleep(100)
            val bndl = Bundle().apply {
                putDouble(MyResultReceiver.DATA_KEY,
                      i * 0.01)
            }
            myReceiver.send(MyResultReceiver.RESULT_CODE, bndl)
        }
    }
}
```

Here is the test class, again inside the androidTest section of the sources:

```
@RunWith(AndroidJUnit4::class)
class MyIntentServiceTest {

    @Test
    fun testIntentService() {
        var serviceVal = 0.0

        val ctx = InstrumentationRegistry.
              getTargetContext()
        val serviceIntent = Intent(ctx,
              MyIntentService::class.java
        ).apply {
            `package`= ctx.packageName
            putExtra(
                MyIntentService.MyResultReceiver.
                      INTENT_KEY,
                MyIntentService.MyResultReceiver( { d->
                        serviceVal = d
                }))
        }
        ctx.startService(serviceIntent)

        val tm0 = System.currentTimeMillis() / 1000
        var ok = false
        while(System.currentTimeMillis() / 1000 - tm0
              < 20) {

        if(serviceVal == 1.0) {
            ok = true
            break
```

```
      }
      Thread.sleep(1000)
    }

    assertThat(ok, `is`(true))
  }
}
```

This test calls the service, listens to its result for a while, and when it detects that the service did its work as expected, lets the test pass.

Testing Content Providers

For testing content providers, Android provides for a special class called `ProviderTestCase2` that starts an isolated temporary environment so the testing won't interfere with a user's data. A test case, for example, reads as follows:

```
@RunWith(AndroidJUnit4::class)
class MyContentProviderTest :
    ProviderTestCase2<MyContentProvider>(
    MyContentProvider::class.java,
    "com.example.database.provider.MyContentProvider") {

  @Before public override   // "public" necessary!
  fun setUp() {
      context = InstrumentationRegistry.
            getTargetContext()
      super.setUp()

      val mockRslv: ContentResolver = mockContentResolver
      mockRslv.delete(MyContentProvider.CONTENT_URI,
            "1=1", arrayOf())
  }

  @Test
  fun test_inserted() {
      val mockCtx: Context = mockContext
      val mockRslv: ContentResolver = mockContentResolver

      // add an entry
      val cv = ContentValues()
      cv.put(MyContentProvider.COLUMN_PRODUCTNAME,
            "Milk")
      cv.put(MyContentProvider.COLUMN_QUANTITY,
            27)
      val newItem = mockRslv.insert(
            MyContentProvider.CONTENT_URI, cv)

      // query all
      val cursor = mockRslv.query(
            MyContentProvider.CONTENT_URI,
```

```
            null, null, null)
      assertThat(cursor.count, `is`(1))

      cursor.moveToFirst()
      val ind = cursor.getColumnIndex(
            MyContentProvider.COLUMN_PRODUCTNAME)
      assertThat(cursor.getString(ind), `is`("Milk"))
  }
}
```

Column names, the authority, and the URI used are biased by the content provider. Important for the test case is that you use the mocked content resolver for talking to the content provider.

> **Note** Observe the order of the first two lines in `setUp()`. This is different from what you can read in the Android developer docs from May 2018. The docs are wrong here.

Testing Broadcast Receivers

For testing broadcast receivers, the Android testing framework does not pay particular attention. It is also crucial what the broadcast receiver under test actually does. Provided it performs some kind of side effect, for example writing something to a database, you can mock out that database operation by using the same testing context we used earlier for content providers.

For example, if you look at the following test case from inside the `androidTest` source section:

```
import android.support.test.InstrumentationRegistry
import android.support.test.runner.AndroidJUnit4
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.Assert.*
import org.hamcrest.Matchers.*
import android.content.Intent

@RunWith(AndroidJUnit4::class)
class BroadcastTest {
  @Test
  fun testBroadcastReceiver() {
      val context = InstrumentationRegistry.
            getTargetContext()

      val intent = Intent(context,
            MyReceiver::class.java)
      intent.putExtra("data", "Hello World!")
      context.sendBroadcast(intent)

      // Give the receiver some time to do its work
      Thread.sleep(5000)
```

```
        // Check the DB for the entry added
        // by the broadcast receiver
        val db = MyDBHandler(context)
        val p = db.findProduct("Milk")
        assertThat(p, isA(Product::class.java))
        assertThat(p!!.productName, `is`("Milk"))
    }
}
```

you can see that we used the context provided by `InstrumentationRegistry.getTargetContext()`. This will make sure the database used by the broadcast receiver and later by the test uses a temporary space for its data.

You start this test like any other integration test by right-clicking it or the package it resides in and then selecting Run or Run Tests in.

User Interface Tests

Conducting user interface tests, you can work through user stories and see whether your app as a whole acts as expected. These are two frameworks:

- **Espresso**

  Use Espresso to write tests targeting your app, disregarding any interapp activities. With Espresso you can do things such as when a certain `View` (`Button`, `TextView`, `EditText`, and so on) shows up, do something (enter text, perform a click), and then you can check whether some postconditions occur.

- **UI Automator**

  Use UI Automator to write tests that span several apps. With UI Automator you can inspect layouts to find out the UI structure of activities, simulate actions on activities, and do checks on UI elements.

For details on how to use either of them, please consult the online documentation. For example, enter *android automating ui tests* in your favorite search engine to find resources.

Chapter 15

Troubleshooting

In the previous chapter, we talked about ways to test your app. If tests fail, the logs usually tell you what exactly happens, and if this is not enough, you can extend the logging of your app to see where things went wrong.

But even with the best possible testing concept, it might still happen that your app doesn't exactly behave as it is supposed to. First, it might just sometimes not do the right things from a functional perspective. Second, it might be ill-behaving from a nonfunctional perspective, which means it eats up memory resources as time goes by, or it might perform badly in terms of speed.

In this chapter, we talk about techniques to remedy problems that your app might expose. We will talk about logging, debugging, and monitoring, as well as the tools in Android Studio and the SDK that help us with respect to those topics.

Logging

Logging in Android is easy; you just import `android.util.Log`, and inside your code you write statements like `Log.e("LOG", "Message")` to issue logging messages. Android Studio then helps you to gather, filter, and analyze the logging.

Although for developing using this logging is extremely handy, when it comes to publishing your app, it gets problematic. You don't want to thwart the performance of your app, and the documentation suggests removing all logging, basically negating all the work you put into the logging. If your users later report problems, you add logging statements for troubleshooting, remove them later again after the fix has been done, and so on.

To rectify this procedure, I suggest instead adding a simple wrapper around the logging from the beginning.

```
class Log {
  companion object {
      fun v(tag: String, msg: String) {
          randroid.util.Log.v(tag, msg)
      }
```


P. Späth, *Pro Android with Kotlin,* https://doi.org/10.1007/978-1-4842-3820-2_15

```
        fun v(tag: String, msg: String, tr: Throwable) {
            android.util.Log.v(tag, msg, tr)
        }

        fun d(tag: String, msg: String) {
            android.util.Log.d(tag, msg)
        }

        fun d(tag: String, msg: String, tr: Throwable) {
            android.util.Log.d(tag, msg, tr)
        }

        fun i(tag: String, msg: String) {
            android.util.Log.i(tag, msg)
        }

        fun i(tag: String, msg: String, tr: Throwable) {
            android.util.Log.i(tag, msg, tr)
        }

        fun w(tag: String, msg: String) {
            android.util.Log.w(tag, msg)
        }

        fun w(tag: String, msg: String, tr: Throwable) {
            android.util.Log.w(tag, msg, tr)
        }

        fun w(tag: String, tr: Throwable) {
            android.util.Log.w(tag, tr)
        }

        fun e(tag: String, msg: String) {
            android.util.Log.e(tag, msg)
        }

        fun e(tag: String, msg: String, tr: Throwable) {
            android.util.Log.e(tag, msg, tr)
        }
    }
}
```

You can then use the same simple logging notation as for the Android standard, but you are free later to change the logging implementation without touching the rest of your code. You could, for example, add a simple switch as follows:

```
class Log {
  companion object {
      val ENABLED = true

      fun v(tag: String, msg: String) {
          if(!ENABLED) return
```

```
            // <- add this to all the other statements
            android.util.Log.v(tag, msg)
        }
        ...
    }
}
```

Or, you could enable logging only for virtual devices. Unfortunately, there is no easy and reliable way to find out whether your app is running on a virtual device. All the solutions presented in blogs have their pros and cons and are subject to change for new Android versions. What you could do instead is to transport build variables to your app. To do so, add the following in your app's `build.gradle` file:

```
buildTypes {
    release {
        ...
        buildConfigField "boolean", "LOG", "false"
    }
    debug {
        ...
        buildConfigField "boolean", "LOG", "true"
    }
}
```

All you have to do then in your logging implementation is replace this:

```
val ENABLED = BuildConfig.LOG
```

which switches on the logging for debugging APKs and otherwise turns it off.

Using a completely different logging implementation is possible as well. For example, to switch the logging to Log4j, add the following inside the `dependencies` section of your app's `build.gradle` file (removing the line breaks after `implementation`):

```
implementation
  'de.mindpipe.android:android-logging-log4j:1.0.3'
implementation
  'log4j:log4j:1.2.17'
```

To actually configure the logging, add the following inside your custom `Log` class:

```
companion object {
  ...
  private val mLogConfigrator = LogConfigurator().apply {
      fileName = Environment.
          getExternalStorageDirectory().toString() +
          "/" + "log4j.log"
      maxFileSize = (1024 * 1024).toLong()
      filePattern = "%d - [%c] - %p : %m%n"
      maxBackupSize = 10
      isUseLogCatAppender = true
      configure()
  }
```

```
  private var ENABLED = true // or, see above
  // private var ENABLED = BuildConfig.LOG

  fun v(tag: String, msg: String) {
      if(!ENABLED) return
      Logger.getLogger(tag).trace(msg)
      // <- add similar lines to all the other
      // statements
  }
  ...
}
```

This example writes the logs to the directory returned by `Environment.getExternalStorageDirectory()`, which on the device usually maps to `/sdcard`. You could do this in other places as well. If you use external storage as shown here, don't forget to check and possibly acquire the appropriate write permissions! More precisely, you need the following in your `AndroidManifest.xml` file:

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

Once your app starts logging to a file inside the device, you can easily access the log file from inside Android Studio by using the file explorer. Start it via View ➤ Tool Windows ➤ Device File Explorer. You can then open the log file by double-clicking it, as shown in Figure 15-1.

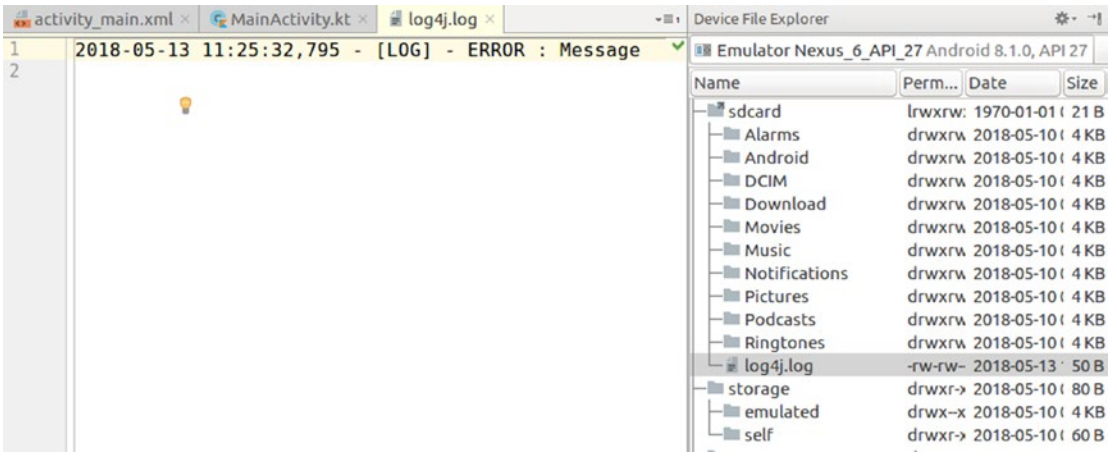

***Figure 15-1.** Accessing log files on the device*

One final measure you could take to improve performance is to use lambdas for logging activities. For this to work, use logging methods as follows in your custom logger:

```
fun v(tag: String, msg: ()->String) {
      if(!ENABLED) return
      Logger.getLogger(tag).trace(msg.invoke())
}
... similar for the other statements
```

Inside your code, you then issue log messages as follows:

```
Log.v("LOG",
      {-> "Number of items added = " + calculate()})
```

The advantage of this approach is that the logging message will not be calculated if logging is not enabled, adding some performance boost to production versions of your app.

Debugging

There is not much to say about debugging from inside Android Studio; it just works as expected.

You set breakpoints inside your code, and once the program flow reaches a breakpoint, you can step through the rest of the program and observe what the program does and how variables change their values.

Performance Monitoring

Android Studio has a quite powerful performance monitor that lets you analyze performance matters down to the method level. To use it, you must first find a way to run that part of your code that is subject to performance issues inside a loop. You can try to use tests for that, but temporarily adding artificial loops to your code is feasible as well.

Then, with that loop running, inside Android Studio open View ➤ Tool Windows ➤ Android Profiler. The profiler first complains that advanced profiling is not enabled, as shown in Figure 15-2.

***Figure 15-2.** Advanced profiling alert*

Enable it by clicking the blue Run Configuration link. Make sure the box is selected, as shown in Figure 15-3, and then click OK.

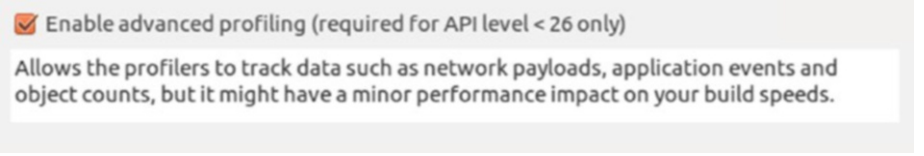

***Figure 15-3.** Advanced profiling setting*

The profiler monitor then shows up, as shown in Figure 15-4. In addition to CPU profiling, it contains memory usage profiling and a network monitor.

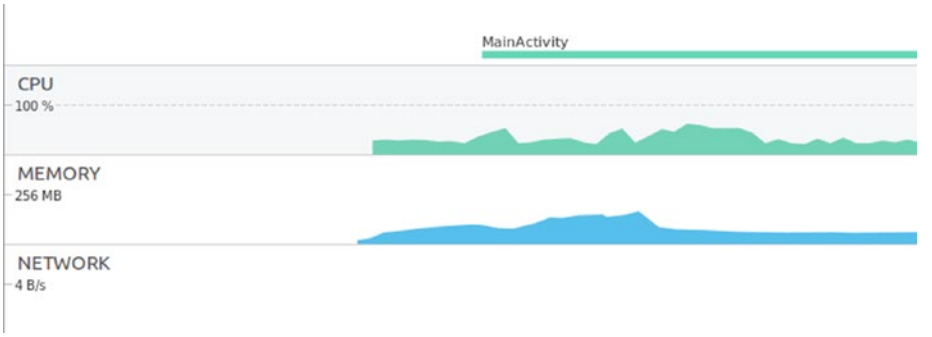

Figure 15-4. Profiler lanes

There clicking the CPU lane narrows the view to the performance monitor diagram you see in Figure 15-5.

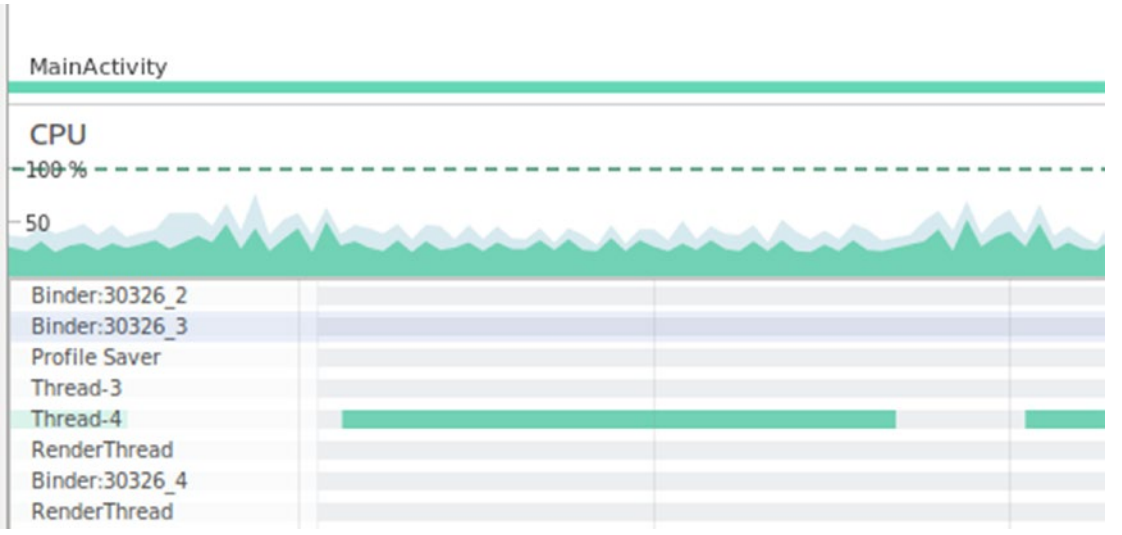

Figure 15-5. The CPU profiling section

Scrolling through the threads in the lower pane, you can then try to find suspicious threads. For the example I am running here, you can see that Thread-4 does quite a lot of work. Renaming it to PiCalcThread (the app calculates pi) and then clicking it shows a message that no data has been captured yet, as shown in Figure 15-6.

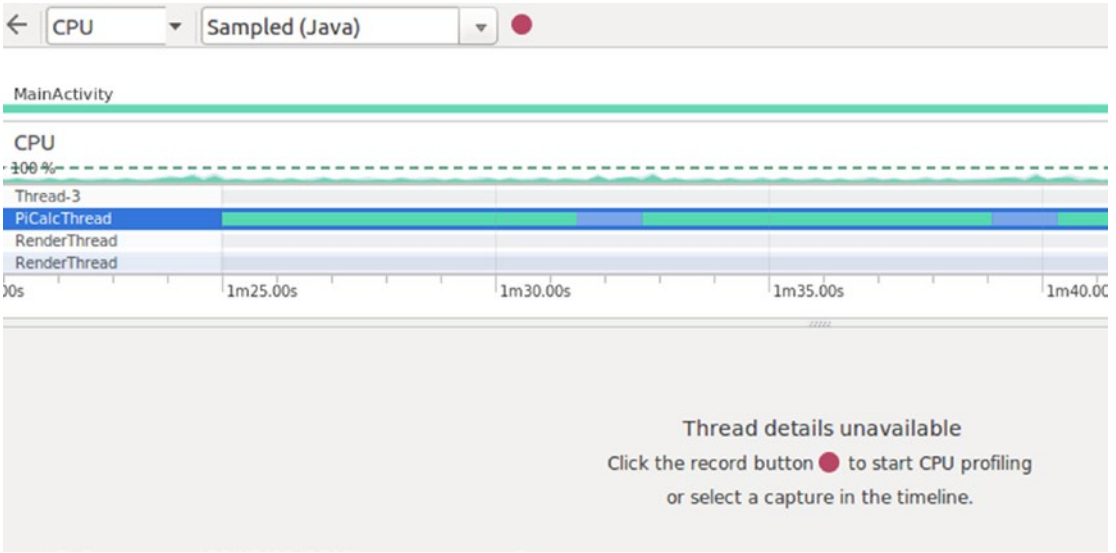

Figure 15-6. *CPU profiling a thread*

On top of the pane, you can see the capturing control, as shown in Figure 15-7.

Figure 15-7. *The CPU profiling capture control*

For the capturing that we are going to start soon, you can choose from these options:

- *Sampled (Java)*: Select this to capture the app's call stack based on a regular interval. This is the least invasive way of capturing, and you will usually choose this.
- *Instrumented (Java):* Select this to collect data for *each and every* method call inside your app. This will introduce a high-performance impact by itself and will collect a lot of data. Choose this if the Sampled variant does not give you enough information.
- *Sampled (Native)*: This is available only on devices starting with Android 8 (API level 26). It will sample native calls. This goes deep into the internals of Android, and you will usually use this only for a deep analysis.

Once you've chosen your capturing mode, start the capturing by clicking the red ball. Let the capturing run for a while and then end it and start analyzing the collected data. Android Studio provides you with different views of the collected data for each thread, and each has its own merit. See Figure 15-8 for a *flame chart*, and see Figure 15-9 for a *top-down chart*.

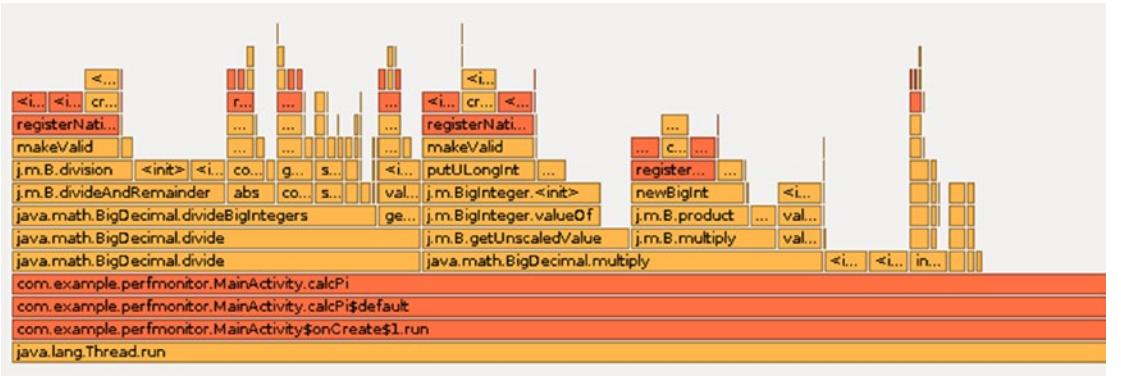

Figure 15-8. A flame chart

Name	Total (µs)	%
PiCalcThread() ()	12,454,599	100.00
run() (java.lang.Thread)	12,454,597	100.00
run() (com.example.perfmonitor.MainActivity$onCreate$1)	12,454,596	100.00
calcPi$default() (com.example.perfmonitor.MainActivity)	12,454,595	100.00
calcPi() (com.example.perfmonitor.MainActivity)	12,454,594	100.00
divide() (java.math.BigDecimal)	4,664,916	37.46
divide() (java.math.BigDecimal)	4,664,915	37.46
divideBigIntegers() (java.math.BigDecimal)	4,169,726	33.48
divideAndRemainder() (java.math.BigInteger)	2,453,961	19.70
abs() (java.math.BigInteger)	563,523	4.52

Figure 15-9. A top-down chart

For this example, scanning through the charts shows you that a considerable amount of the CPU power gets spent in `BigDecimal.divide()`. To improve the performance for this example, you could try to avoid calling this method too often, or you could try to find a substitute.

As an extra aid for the analysis, you can switch on a filter. Click the filter symbol on the right of the controller pane, as shown in Figure 15-10. Android Studio then highlights matching entries inside the charts, as shown in Figure 15-11.

Figure 15-10. The profiling filter

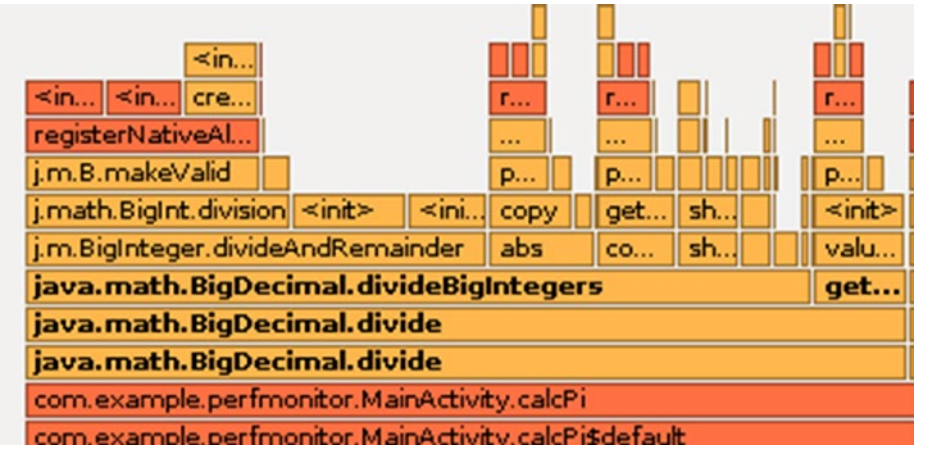

Figure 15-11. The profiling filter switched on

For more information and details about performance monitoring, please see Android Studio's documentation.

Memory Usage Monitoring

In addition to profiling the app's performance, as shown in the previous chapter, the Android Studio's profiler helps you find memory leaks or issues related to poor memory management. Again, put the parts of the code subject to problems into a loop and start it. Open the profiler via View ➤ Tool Windows ➤ Android Profiler. Choose the memory lane, and immediately the profiler shows you a memory usage diagram, as shown in Figure 15-12.

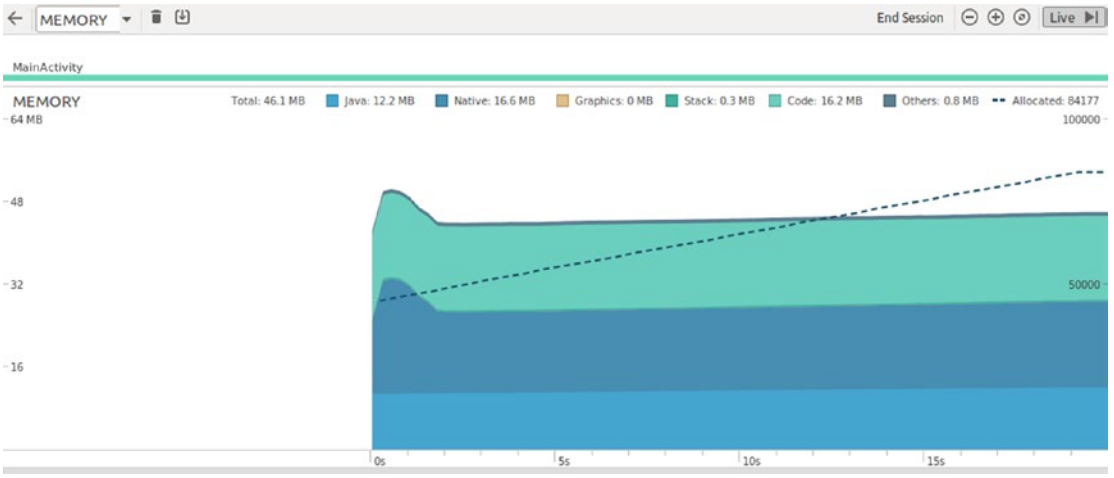

Figure 15-12. The Memory Monitor

After this runs for a while, you can see that the memory usage rises. This is because in the sample app I added an artificial memory leak. See Figure 15-13.

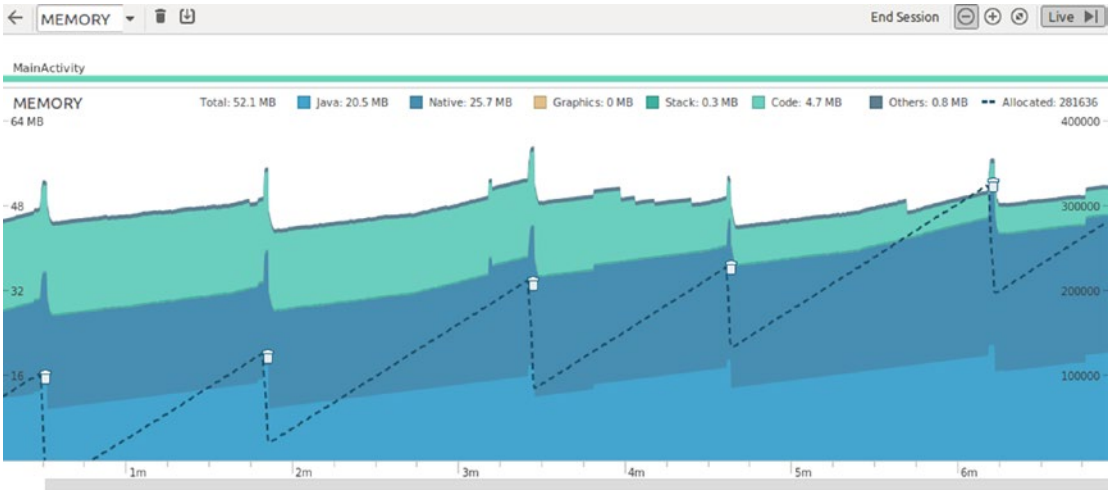

Figure 15-13. Memory profiling, longer period

To start an analysis, select an appropriate region using the mouse. The view then immediately switches to a usage statistics view, as shown in Figure 15-14.

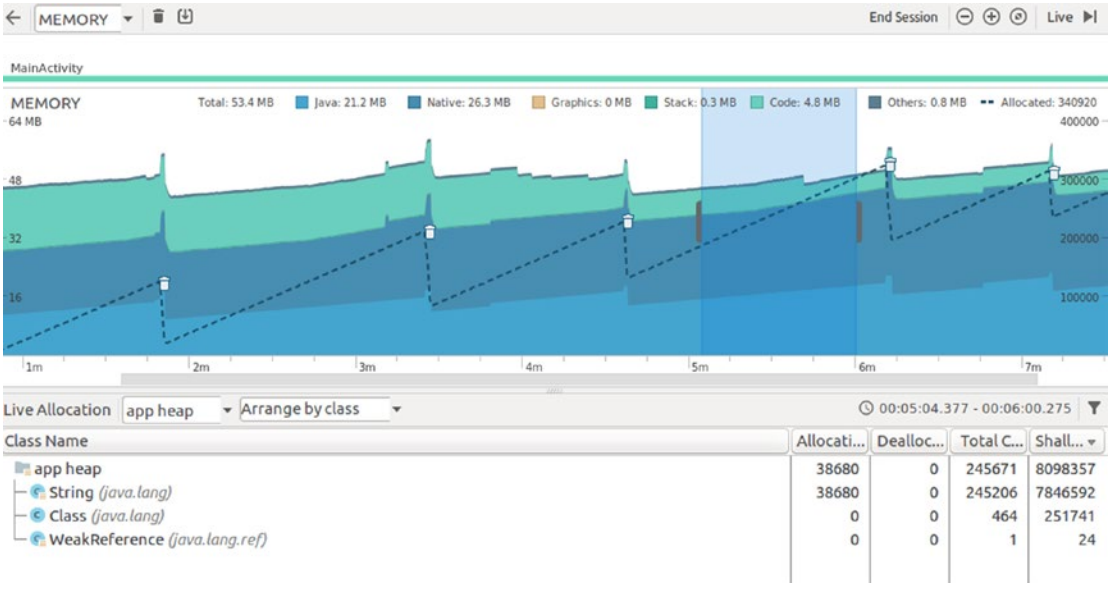

Figure 15-14. Memory profiling, selected

To find the leak, switch from the "Arrange by class" mode to "Arrange by callstack." Dive into the tree by clicking, double-clicking, and/or pressing the Enter key. The final result might look like Figure 15-15.

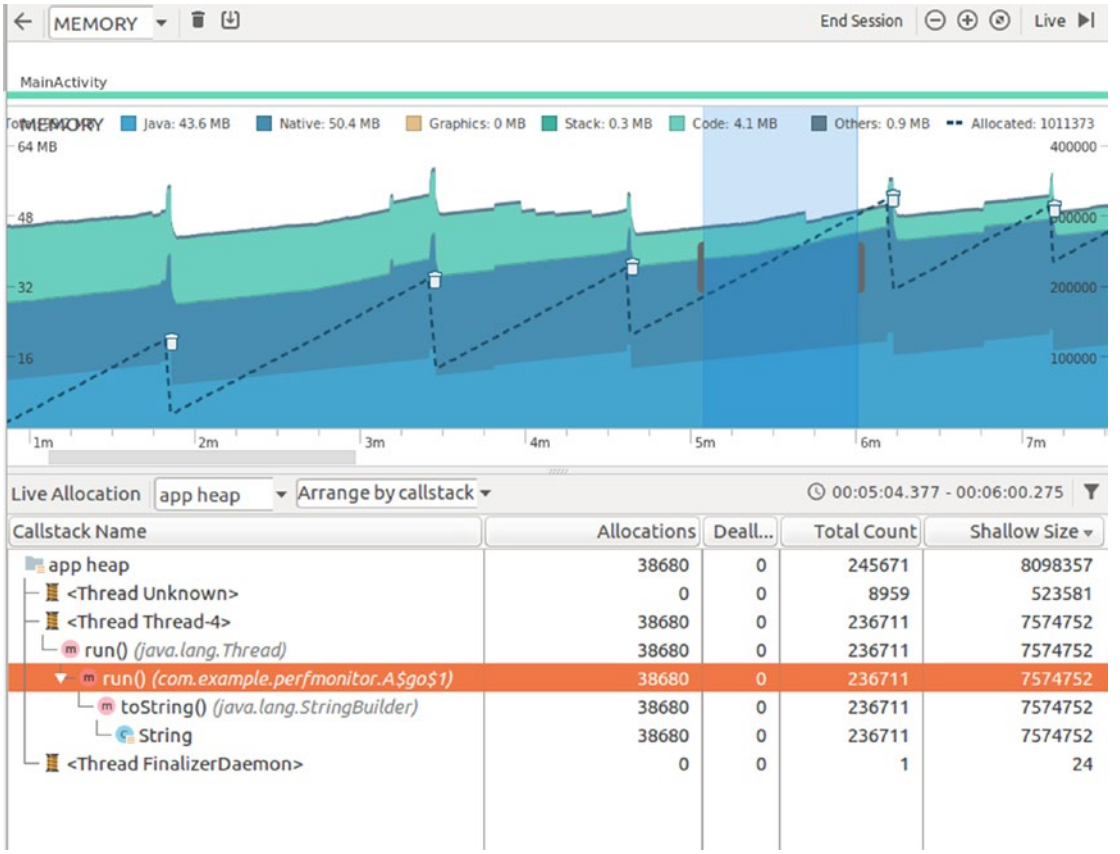

Figure 15-15. *Memory profiling, details*

The orange line with almost 40,000 allocations belongs to `run() (com.example.perfmonitor.A$go$1`, which is exactly the point where I've put the memory leak.

```
class A {
  fun go(l:MutableList<String>) {
      Thread {
          while (true) {
              l.add("" + System.currentTimeMillis())
              Thread.sleep(1)
          }
      }.start()
  }
}
```

If this is not enough to troubleshoot memory problems, you can acquire a heap dump. To do so, click the Heap Dump symbol in the header of the profiler window, as shown in Figure 15-16.

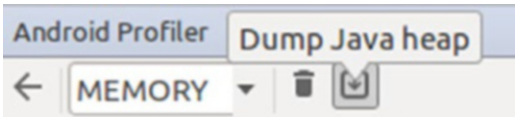

Figure 15-16. *Taking a heap dump*

You can then use the same techniques as described earlier or export the heap as an HPROF file and use other tools to analyze the dump. To perform such an export, click the Export icon in the left-top corner of the Heap Dump view, as shown in Figure 15-17.

Figure 15-17. *Saving a heap dump*

> **Note** Such a heap dump allows you to determine object reference relationships—something that goes beyond the memory analysis of Android Studio. This gives you the maximum insight into the memory structure, but it takes some time to get acquainted with heap dump analysis tools and to find the correct answers.

Chapter 16

Distributing Apps

If you have finished your app, you need to find a way to distribute it. The primary place where to go for that purpose is the Google Play store, but it is also possible to use other distribution channels if you can convince your users to allow app installations from "other sources." I do not present a list of distribution channels here, nor do I present detailed instructions for using the Google Play store. There are just too many options depending on which market you are targeting. Also, this book is not intended to be an introduction into app marketing in general.

Your Own App Store

Now that devices allow users to install apps from sources other than the Google Play store, APK files can be presented from any server including your own corporate servers. Note that the process is different depending on the Android version used.

- Up to Android 7 (API level 25), there is a system-wide setting in the "security" section to allow app installation from other sources than Google Play.
- Starting with Android 8 (API level 26), the permission to install apps from other sources is handled on a per-app basis, for example a setting in the browser.

No matter which distribution channel you choose, you must first generate a signed APK via Build ➤ Generate Signed APK. Then, copy it to the server and make sure the file gets assigned the MIME type `application/vnd.android.package-archive`.

> **Note** Although Android Studio automatically uploads debug versions of your app to virtual devices or devices connected by USB, for virtual devices you can also test the signed APK installation procedure. If you have a server running on your local development machine, inside the virtual device use the IP 10.0.2.2 to connect to the development machine. Better first uninstall versions installed by the development build process.

P. Späth, *Pro Android with Kotlin,* https://doi.org/10.1007/978-1-4842-3820-2_16

You can then use the device's browser to download and install APK files.

The Google Play Store

Despite this not being an introduction into how to use the Google Play store, here are a couple of additional points for the technical aspects of distribution:

- As stated, you *must* sign your app before it can be distributed to Google Play.
- The online documentation suggests removing all logging statements from inside your app prior to distributing it. As a less destructive alternative, follow the instructions from Chapter 15 and create a custom logger.
- If your app uses a database, provide for update mechanisms when the database schema changes. See the class `SQLiteOpenHelper`. If you forget that, updating apps and upgrading the database from version to version can become really cumbersome.
- It is possible to distribute different APKs for different devices. This feature was somewhat neglected in this book, because nowadays with modern devices the size of an app no longer plays a vital role, and you usually can put everything into a single APK file. If you still want to provide multiple APKs, please consult the online documentation. Search for *android multiple apk* or similar using your favorite search engine.
- If you test your app on a real device, things become a little easier if on your device you use a different Google account compared to the account you use for distributing the app. Google otherwise won't let you install your app using the Play store. Do this early because changing the Google account of a device later might be complicated.
- Localize all text shown to the user! In this book, localization was not used for brevity reasons, but you definitely should do that for your app. The LINT checker included with Android Studio helps find localization deficiencies, and using the custom locale switcher included with the emulators lets you do a check as well.
- Although developing just for a smartphone form factor (screen size, resolution) is somewhat tempting, you should check your design for other form factors. The various emulators help you with that. You should at least also test your app for a tablet.

Chapter 17

Instant Apps

Instant apps allow a device user to use apps without actually installing them. On the Google Play store, the “Try it” button you sometimes see starts such an instant app.

DevelopingInstant Apps

To develop instant apps, there is a switch you can use while creating an Android app, as shown in Figure 17-1.

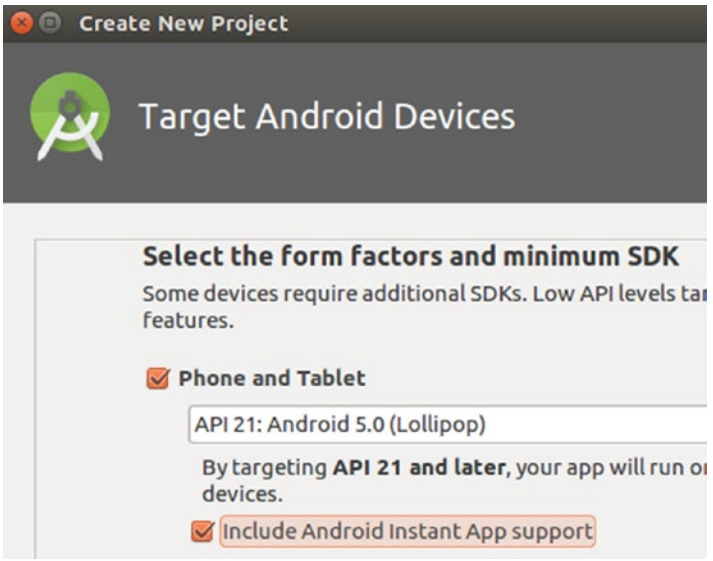

Figure 17-1. *Adding instant app features*

P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_17

Say you named the project `instantapp`; the wizard creates four modules, as shown here:

- **base**

  Contains the basis for both the normal installable app variant and the instant app. In contrast to what many blogs suggest, you need not put anything important here. Android Studio creates just two files inside this module: a `build.gradle` file that as its main characteristics contains the marker `baseFeature true` and a very basic `AndroidManifest.xml` file you don't need to adjust while adding components and code. By virtue of the build file, the base module depends on both the installable app and the instant app variant.

Note: For clean design purists, both the installable app and the instant app depend on the base module in turn, which smells like a circular dependency. The dependency is not to be understood as a Java package dependency, though!

- **app**

  Contains the build instructions for the installable app. This does not contain code since both the installable app and the instant app share the same code basis, which goes into the `feature` module.

- **instantapp**

  Contains the build instructions for the instant app. This does not contain code either.

- **feature**

  The shared code for the installable app and the instant app goes here.

As of May 2018, there is a mismatch between the wizard's output and what the Google Play store expects. To avoid problems later when you want to roll out your instant app, change the `AndroidManifest.xml` file of the `feature` module and add another `<data>` element for the `http` scheme as follows:

```
<data
    android:host="myfirstinstantapp.your server.com"
    android:pathPattern="/instapp"
    android:scheme="https"/>
<data
    android:scheme="http"/>
```

Also, the intent filters must have the attribute `android:autoVerify = "true"` added. The Play store will check for it and complain if it is missing.

The rest of the development does not substantially differ from normal Android app development. Just running the instant app is different from what you know. We will talk about that in the following sections.

Testing Instant Apps on an Emulator

Instant apps can be tested on an emulator. To do so, make sure the run configuration chosen shows `instantapp`, as shown in Figure 17-2.

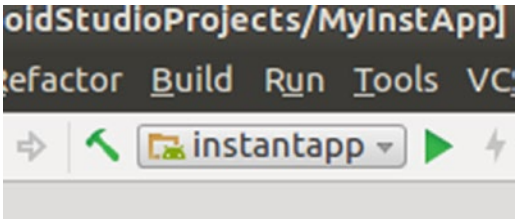

Figure 17-2. Run configuration

Also, if you open Edit Configuration from inside the menu that pops up when you press the small gray triangle, you should see the URL launch method selected, as shown in Figure 17-3.

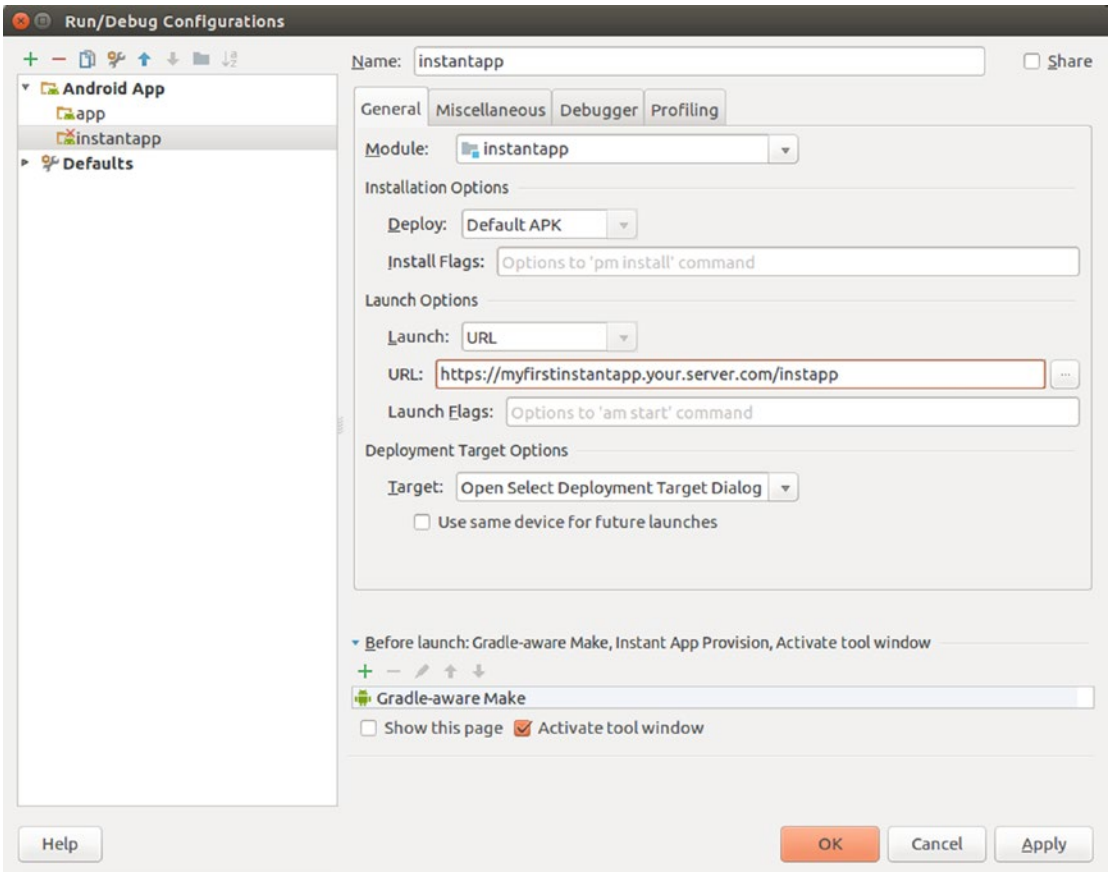

Figure 17-3. Launch method

For running on an emulated device, it doesn't matter whether the URL entered exists, but it must match the host specification inside the intent filter from `AndroidManifest.xml` of the module `feature`. Otherwise, the configuration screen will complain.

> **Caution** For development, adding an `android:port` attribute will lead to problems. Depending on your circumstances, you might need one later when you want to roll out your app, but during development don't use one, or comment it out!

Building Deployment Artifacts

Before an instant app can be rolled out, you must build signed APKs for *both* the installable app and the instant app.

> **Caution** Both variants and also the base module need to have the same version info as shown inside their `build.gradle` files.

To create the deployment artifacts, go to Build ➤ Generate signed APK twice, for both `app` and `instantapp`.

The deployment artifact for the installable app is as usual an `.apk` file, and for the instant app it is a zip file.

Preparing Deep Links

Deep links are URLs that show up in web pages or apps and are linked to features of instant apps. Whenever a user clicks or taps a URL that by virtue of the intent filters of a rolled-out instant app maps to it, the corresponding feature gets downloaded and started immediately without the need to install it.

For production apps, the URLs connected to instant apps must exist, and the domain must have at its root a file called `.well-known/assetlinks.json`. By the way, Google verifies that the domain you are referring to exists and is yours. The structure of this file gets explained in the online documentation, but Android Studio also has a wizard for it: go to Tools ➤ App Links assistant.

If you generate the file `assetlinks.json` manually, you need to enter the certificate fingerprint. Unless you already have it, you can get it via the following:

```
keytool -list -v -keystore my-release-key.keystore
```

An example for such a file is as follows, with the fingerprint cropped:

```
[{
  "relation":
      ["delegate_permission/common.handle_all_urls"],
  "target": {
    "namespace": "android_app",
    "package_name": "com.example",
    "sha256_cert_fingerprints":
        ["14:6D:E9:83:C5:73:06...50"]
  }
}]
```

Rolling Out Instant Apps

To roll out instant apps with the Google Play console, you must create a new app and then roll out both the installable app and the instant app.

During the process, the Play console will perform various checks to see if everything is configured the right way inside your app, and it will also check whether your server was set up correctly as described in the previous section.

Chapter 18

CLI

In this chapter, we summarize the command-line tools that you can use for building, administering, and maintaining tasks running outside Android Studio.

Note

- These tools in part are kind of "semi" official; you might find them at slightly different locations inside the SDK folder, and the online documentation is not necessarily up-to-date.
- The tools shown here are part of the Linux distribution. For other OSs, counterparts are provided, and their usage will be similar.
- The list is not exhaustive; it might differ from your installation if you have fewer or more SDK packages installed.

The SDK Tools

Table 18-1 describes the platform-independent tools provided in this folder:

```
SDK_INST/tools/bin
```


P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2_18

Table 18-1. SDK Tools

Command	Description
`apkanalyzer`	Use this to analyze the APK files that you can find, for example, in the `PROJECT-DIR/PROJECT/release` folder (`PROJECT` quite often reads `app`). Invoking the command without an argument, as shown here, displays usage information: `./apkanalyzer`
`archquery`	This is a simple tool to query your OS's architecture. `./archquery` This outputs, for example, `x86_64`.
`avdmanager`	Use this to manage virtual devices (AVD = Android virtual devices). You can create, move, and delete devices, and you can list devices, targets, and AVDs. Invoking the command without an argument, as shown here, displays usage information: `./avdmanager` You can find the data files for virtual devices handled by this command in `~/.android/avd`. The system images used for creating devices are in `SDK_INST/system-images`.
`jobb`	Use this to manage Opaque Binary Blob (OBB) files. Those are *APK expansion files* that go on an external storage, for example an SD card, and are accessible only from inside your app. Invoking the command without an argument, as shown here, displays usage information: `./job`
`lint`	This is the LINT tool for code inspection. Invoking the command without an argument, as shown here, displays usage information. `./lint`
`monkeyrunner`	This is a powerful testing tool for controlling Android apps by use of a Python script on your PC. Invoking the following shows usage information: `./monkeyrunner` Starting it without an argument launches a Jython shell. You can find more details about `monkeyrunner` in Chapter 14.
`screenshot2`	Use this to take a screenshot from devices or emulators. Invoking the command without an argument, as shown here, displays usage information: `./screenshot2`

(*continued*)

Table 18-1. (*continued*)

Command	Description

`sdkmanager`	This tool helps you manage packages for the Android SDK. You can install, uninstall, or update SDK packages, and you can use it to list installed and available packages. Invoking
	`./sdkmanager --help`
	shows verbose usage information. For example, to list installed and available packages, including build tools, platforms, documentation, sources, system images, and more SDK components, invoke this:
	`./sdkmanager --list`
	To install new components, the tool needs to download them. A couple of flags exist; see the output of `-help` to find out how to specify proxies or disable the usage of HTTPS.

`uiautomatorviewer`	This opens the UI Automator GUI. `./uiautomatorviewer` See Chapter 14 for more information.

The tools focus on the management of virtual devices, the SDK itself, and various testing and artifact management tasks.

In the parent directory, shown here:

```
SDK_INST/tools
```

you will find a couple of more tools. See Table 18-2 for a summary of them.

Table 18-2. More SDK Tools

Command	Description
`android`	Deprecated. Invoke it without an argument to see a synopsis.
`emulator`	The emulator management tool. We talked about the emulator in Chapter 1. Invoke `./emulator -help` to get usage information for this command.
`emulator-check`	A diagnosis tool for the host system. See the output of `./emulator-check -h` for a synopsis.
`mksdcard`	Creates a FAT32 image to be used as an emulator image. Invoke it without an argument for usage information, as shown here: `./mksdcard`

(*continued*)

Table 18-2. (*continued*)

Command	Description
`monitor`	Starts the graphical device monitor. This is the same device monitor as the one invoked from inside Android Studio at Tools ➤ Android ➤ Android Device Monitor. Note that if you run this command while an instance of Android Studio is running, you might get an error message.
`proguard`	Inside this directory the Proguard program resides. With Proguard you can shrink APK files by disregarding files, classes, fields, and methods. Find "Shrink Your Code and Resources" inside the online documentation to learn how Proguard works.

The SDK Build Tools

Table 18-3 lists the build tools provided inside this folder:

`SDK_INST/build-tools/[VERSION]`

Table 18-3. SDK Tools

Command	Description
`aapt`	This is the Android asset packaging tool. Invoking the command without an argument, as shown here, displays usage information: `./aapt` The tool is able to list the contents of an APK file, and it can extract information from it. Furthermore, it is able to package assets, add to, and remove elements from an APK file. The tool is also responsible for creating the R class, which maps resources to resource IDs usable from inside the code (Android Studio does that automatically for you).
`aapt2`	This is the successor of the `aapt` tool described earlier. Invoking the command without an argument, as shown here, displays some basic usage information: `./aapt2` Invoking any of `./aapt2 CMD -h` with `CMD` one of `compile`, `link`, `dump`, `diff`, `optimize`, or `version` gives more detailed information. Cross-checking with the help of the `aapt` command gives extra aid.
`aarch64-linux-android-ld`	A special linker for Android object files, targeting at devices with a 64-bit ARM architecture. Invoking the command displays verbose usage information, as shown here: `./aarch64-linux-android-ld --help` Normally you don't have to invoke that tool directly if you use Android Studio because it takes care of both compiling and linking for you.

(*continued*)

Table 18-3. (*continued*)

Command	Description
`aidl`	AIDL is the Android Interface Definition Language handling low-level interprocess communication between *Bound Service* classes of different apps. The `aidl` tool can be used to compile an `*.aidl` interface definition file to the Java language interface files defining the interface. Invoking the command without arguments shows usage information. `./aidl`
`apksigner`	Manages APK file signing. APK files need to be signed before they can be published. Android Studio helps you with that process (see Build ➤ Generate Signed APK), but you can also use this tool. Invoke it as follows for usage information: `./apksigner -h`
`arm-linux-androideabi-ld`	A special linker for Android object files, targeting at devices with 32-bit ARM architecture and for object files that have been generated by the ABI compiler. Invoking the command shows verbose usage information. `./arm-linux-androideabi-ld --help` Normally you don't have to invoke that tool directly if you use Android Studio, because it takes care of both compiling and linking for you.
`bcc_compat`	A BCC compiler used for `renderscript` by Android. Invoking the command as follows shows usage information: `./bcc_compat --help`
`dexdump`	A tool for investigating DEX files that live inside an APK file and contain the classes. Invoking without arguments, as shown here, displays usage information: `./dexdump`
`dx`	A tool for managing DEX files. You can, for example, create DEX files or dump their contents. Invoke the following to get usage information: `./dx --help`
`i686-linux-android-ld`	A linker for Android object files, targeting devices with an x86 architecture. Invoking the command, as shown here, displays verbose usage information: `./i686-linux-android-ld --help` Normally you don't have to invoke that tool directly if you use Android Studio because it takes care of both compiling and linking for you.

(*continued*)

Table 18-3. (*continued*)

Command	Description
`llvm-rs-cc`	The renderscript source compiler (offline mode). Invoke `./llvm-rs-cc --help` to see some usage information.
`mainDexClasses`	This is for legacy application wanting to allow – `multi-dex` on command `dx` and load the multiple files using the `com.android.multidex.installer` library. The `mainDexClasses` script will provide the content of the file to give to `dx` in `–main-dex-list`.
`mipsel-linux-android-ld`	A linker for Android object files, targeting at devices with an MIPS architecture. Invoking the command `./mipsel-linux-android-ld --help` shows verbose usage information. Normally you don't have to invoke that tool directly if you use Android Studio because it takes care of both compiling and linking for you.
`split-select`	Allows for generating the logic for selecting a split APK given a target device configuration. Invoking the command `./split-select --help` shows some usage information.
`x86_64-linux-android-ld`	A linker for Android object files, targeting devices with an x86 64 bit architecture. Invoking the command `./x86_64-linux-android-ld --help` shows verbose usage information. Normally you don't have to invoke that tool directly if you use Android Studio because it takes care of both compiling and linking for you.
`zipalign`	ZIP alignment utility. Something developers are not necessarily used to is the fact that an operating system might depend on elements of an archive file aligned a certain way, for example entries always starting at 32-bit boundaries. This tool can be used to accordingly adapt a ZIP file. Invoking `./zipalign -h` shows usage information.

Contained are linkers, compilers, APK file tools, and an Android Interface Definition Language (AIDL) management tool.

The SDK Platform Tools

Table 18-4 describes the platform-dependent tools provided inside this folder:

```
SDK_INST/platform-tools
```

***Table 18-4.** SDK Platform Tools*

Command	Description
`adb`	The *Android Debug Bridge*. See the text after the table for a description of the `adb` command.
`dmtracedump`	Creates graphical call-stack diagrams from a trace dump. The trace file must have been acquired with the `android.os.Debug` class. Invoke it without arguments, as shown here, to get information about the command: `./dmstracedump`
`e2fsdroid`	Mount an image file. Currently broken.
`etc1tool`	Use this to convert between the PNG and ETC1 image format. Invoke `./etc1tool --help` to see usage information.
`fastboot`	This is the *fastboot* program you can use to modify your device's firmware. Invoke `./fastboot --help` for usage information.
`hprof-conv`	Use this to convert the HPROF heap file you got from the Android OS tools to a standard HPROF format. Invoke it without an argument, as shown here, to get usage information: `./hprof-conv`
`make_f2fs`	Used to make an F2FS filesystem on some device. Invoke it without arguments, as shown here, for usage information: `./make_f2fs`
`mke2fs`	Generates a Linux Second Extended file system. Invoke it without arguments, as shown here, to see options: `./mke2fs`
`sload_f2fs`	Used to load files into a F2FS device. Invoke it without arguments, as shown here, to see options: `./sload_f2fs`
`sqlite3`	Starts a SQLite administration tool. Invoke it as shown here for usage information: `./sqlite3 -help`

(*continued*)

Table 18-4. (*continued*)

Command	Description
`systrace/ systrace.py`	The graphical Systrace utility to investigate the Android system. The path where the tool `adb` resides must be part of the `PATH` environment variable, and you must have Python installed. You can then run `python systrace/systrace.py -h` for a command synopsis.

The Android Debug Bridge (ADB) invoked by the `adb` command is a versatile tool to connect your development PC to running emulators and devices connected via USB or Wi-Fi. It consists of a client and transparent server process on the development PC and a daemon running on the device. You can use `adb` to do the following:

- Query for accessible devices
- Install and uninstall apps (APK files)
- Copy files to or from a device
- Perform backup and restore
- Connect to the logging output of an app
- Enable root access on a device
- Start a shell on a device (to, for example, see and investigate your app's files)
- Start and stop activities and services
- Issue broadcasts
- Start and stop profiling sessions
- Dump heaps
- Access the package manager on the device
- Take screenshots and record videos
- Restart devices.

For more details, find the "Android Debug Bridge" page in the online documentation. Invoke it via the following to show the help provided with the command:

```
./adb
```

For example, use this to list the connected devices:

```
./adb devices
```

To open a shell on a device, use the following, with the `DEVICE_NAME` argument being one of the entries from the first column from the devices list:

```
./adb -s DEVICE_NAME shell
```

If there is only one device, in the previous command you can omit the `-s` flag and the device name.

> **Note** You must enable debugging on real devices for ADB to successfully connect to them.

Index

A

P. Späth, *Pro Android with Kotlin*, https://doi.org/10.1007/978-1-4842-3820-2

B

C

D

E

F

G

H

I

J

K

L

Q

R

S

T

U

V

W, X, Y, Z