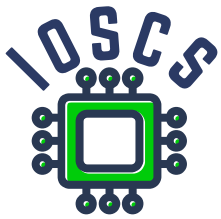
Mendel University in Brno

Mobile Application Development
Study material

Radosław Maciaszczyk
West Pomeranian University of Technology in Szczecin

Project: Innovative Open Source Courses
for Computer Science Curriculum



24. 6. 2022

Co-funded by the Erasmus+ Programme of the European Union

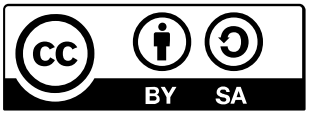
West Pomeranian University of Technology Szczecin

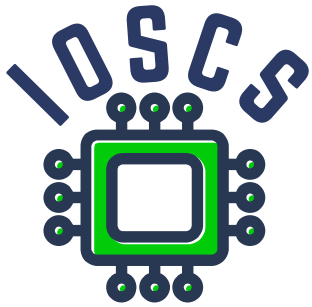
UNIVERSITY OF ŽILINA

Mendel University in Brno

Reviewer: Dr. Patrik Hrkút, Department of Software Technologies,
University of Žilina, Slovakia
Project: Innovative Open Source Courses for Computer Science Curriculum

ISBN 978-80-7509-890-0 (online ; pdf)
DOI: https://doi.org/10.11118/978-80-7509-890-0





This material teaching was written as one of the outputs of the project "Innovative Open Source Courses for Computer Science Curriculum", funded by the Erasmus+ grant no. 2019-1-PL01-KA203-065564. The project is coordinated by West Pomeranian University of Technology in Szczecin (Poland) and is implemented in partnership with Mendel University in Brno (Czech Republic) and University of Žilina (Slovak Republic). The project implementation timeline is September 2019 to December 2022.

Project information

Project was implemented under the Erasmus+.
Project name: **"Innovative Open Source courses for Computer Science curriculum"**
Project nr: **2019-1-PL01-KA203-065564**
Key Action: **KA2 – Cooperation for innovation and the exchange of good practices**
Action Type: **KA203 – Strategic Partnerships for higher education**

Consortium
ZACHODNIOPOMORSKI UNIWERSYTET TECHNOLOGICZNY W SZCZECINIE
MENDELOVA UNIVERZITA V BRNĚ
ŽILINSKÁ UNIVERZITA V ŽILINE

Co-funded by the
Erasmus+ Programme
of the European Union

Preface

The book has been prepared to acquire or extend knowledge in the field of "Mobile Application Development" in the Android environment. Together with the material supporting the labs and lectures, it provides a complete course for those starting to program in the Android environment. The book can be used by undergraduate students, engineering students, graduate students, new technology enthusiasts, and engineers wishing to gain knowledge in programming in the Android environment. Teachers and lecturers can use the entire course along with the book to teach. The course is dedicated to beginners, hence the material included here covers the basics. The book uses official documentation, descriptions, and guides available at `https://android.com`. It should be noted that the Android platform is constantly evolving, adding or changing features or libraries. This results in the fact that some of the material may change over time. However, the topics selected in the book represent a certain base, and their variability is already tiny.

The first two chapters provide general information about Android and the tools for developing applications. The third chapter contains a description of the basic components of Android, together with an overview of their life cycle. This is followed by information on how to create the user interface. The fifth chapter includes details on how sensors and locations are used. Ways of storing data are included in chapter six. Chapter seven familiarises you with the MVVM design pattern. The final chapter provides information on how to retrieve data from the internet.

Contents

Introduction

Technological advances, together with the development of wireless communication systems, are one of the main drivers for the development of operational mobile systems in the consumer market. Another element contributing to the development of systems and devices is social media and e-commerce systems. Mobile devices, in particular smartphones, are increasingly replacing other devices. Devices equipped with powerful and energy-efficient processors, various types of sensors or capacious and fast-charging batteries. This allows them to be used for a variety of activities. The scope of these activities is broad, thanks to the possibility of developing applications. "Digital 2022" reports series [3] in partnership with "We Are Social" and "Hootsuite" published that 66.6% of the world's population uses mobile phones. The same report publishes that the average usage time of the device is 4h 10min, with 92% of the time using various mobile applications. This shows great potential for application development.

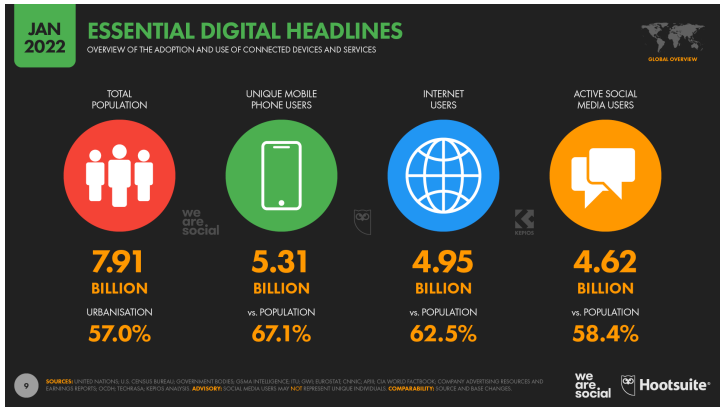


Figure 1.1: Mobile Users [3]

By mobile devices, we mean electronic devices that allow data processing without a wired connection to computer networks. Another important aspect is that their small size or battery power, which will enable users to carry this device around. The market for consumer mobile devices includes smartphones, tablets or smartwatches. The operating systems used in these devices can also be found in other devices such as televisions or cars' info-entertainment systems. However, these recently mentioned devices do not fall into the category of mobile devices. This shows a certain evolution of mobile operating systems, which have become more.

The market is currently dominated by two operating systems (Tab. 1.1), Android and iOS. The high popularity of Android is because this system is used by many manu-

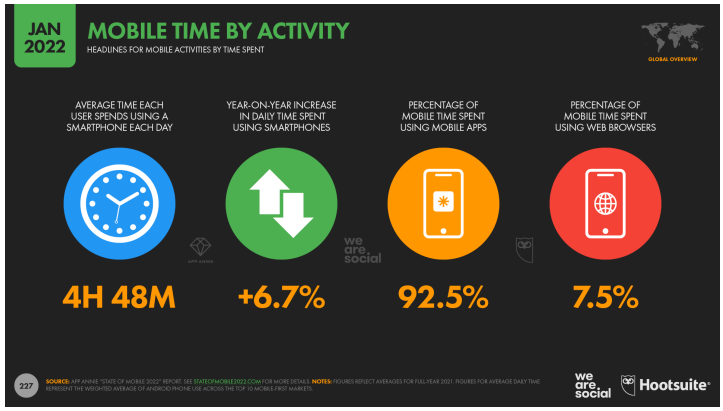


Figure 1.2: Mobile Time by Activity [3]

facturers producing devices with different prices. There are several leading vendors with a predominance of companies from the East Asian regions (Tab. 1.2).

Tab. 1.1: *Mobile Operating System Market Share Worldwide, March 2022 [1]*

Mobile Operating System	*Market share*
Android	71,7%
iOS	27,57%
Samsung	0,42%
KaiOS	0,14%
Unknown	0,1%
Windows	0,01%

Android is an open-source operating system for mobile devices and a related open-source project led by Google [11]. Android Open Source Project (AOSP) repository offers the information and source code needed to create custom variants of the Android OS, port devices and accessories to the Android platform, and ensure devices meet the compatibility requirements that keep the Android ecosystem a healthy and stable environment for millions of users.

As an open-source project, Android's goal is to avoid any central point of failure in which one industry player can restrict or control the innovations of any other player. To that end, Android is a whole, production-quality operating system for consumer products, complete with customisable source code that can be ported to nearly any device and public documentation available to everyone.

There are not many requirements to start developing applications. Generally, you need knowledge of the object-oriented language. For Android, it is (JAVA, KOTLIN).

1.1 History of Android

The Android story began in October 2003 in Palo Alto, California. This is when four men – Andy Rubin, Rich Miner, Nick Sears and Chris White – decide to set up an Android Inc

Tab. 1.2: *Mobile Vendor Market Share Worldwide, March 2022 [2]*

Mobile Operating System	*Market share*
Samsung	28,22%
Apple	27,57%
Xiaomi	12,24%
Huawei	6,53%
Oppo	5,25%
Vivo	4,12%
Realme	2,99%
Motorola	2,66%
LG	1,26%
Other	9,16%

company. Initially, the aim of Android Inc. was software for digital cameras, but over time they changed the profile of the software they were developing to a mobile operating system. This was to compete with Nokia's Symbian or Microsoft's Windows Mobile. In July 2005, Android Inc. was acquired by Google. At the time, the popularity of mobile systems was only growing, and the purchase was one of many at the time and did not generate much interest. In 2007, in response to Apple, Google and 34 partners formed the Open Handset Alliance [4] to develop and popularise the new Android mobile operating system. The alliance brought together several companies from different fields:

- Mobile Operators including NTT DoCoMo,
- Handset Manufacturers including HTC, Samsung,
- Semiconductor Companies including Qualcomm,
- Software Companies including Google, eBay,
- Commercialization Companies including Flex Comix.

Since then, the Open Handset Alliance has been working on Android development. The first version of the Android SDK was released on 12.11.2007 [5], which includes development tools, a debugger, libraries, an emulator, documentation, sample projects, tutorials, FAQs and much more. However, it was not until the commercial launch (22.10.2008) of the first phone [6] with Android that the rapid development of the system began.

In early 2022, version 12 of Android [7] is officially available, along with API version 32. Subsequent versions of Android and their corresponding APIs change the system's appearance and add new capabilities or improvements. The versions of the system released to date are shown in table 1.3. The latest version of Android 13 is the version planned for deployment. This version was in development at the time of writing, and the API was available to developers.

1.2 Android and Open Source

Android is an open-source operating system for mobile devices and a corresponding open-source project led by Google [11]. Android Open Source Project (AOSP) repository

Tab. 1.3: *Android versions [7]*

Name	*Internal codename*	*Version number(s)*	*Initial stablerelease date*
Android 1.0	N/A	1.0	September 23, 2008
Android 1.1	Petit Four	1.1	February 9, 2009
Android Cupcake	Cupcake	1.5	April 27, 2009
Android Donut	Donut	1.6	September 15, 2009
Android Eclair	Eclair	2.0 2.0.1 2.1	October 27, 2009 December 3, 2009 January 11, 2010
Android Froyo	Froyo	2.2 – 2.2.3	May 20, 2010
Android Gingerbread	Gingerbread	2.3 – 2.3.2 2.3.3 – 2.3.7	December 6, 2010 February 9, 2011
Android Honeycomb	Honeycomb	3.0 3.1 3.2 – 3.2.6	February 22, 2011 May 10, 2011 July 15, 2011
Android Ice Cream Sandwich	Ice Cream Sandwich	4.0 – 4.0.2 4.0.3 – 4.0.4	October 18, 2011 December 16, 2011
Android Jelly Bean	Jelly Bean	4.1 – 4.1.2 4.2 – 4.2.2 4.3 – 4.3.1	July 9, 2012 November 13, 2012 July 24, 2013
Android KitKat	Key Lime Pie	4.4 – 4.4.4 4.4W – 4.4W.2	October 31, 2013 June 25, 2014
Android Lollipop	Lemon Meringue Pie	5.0 – 5.0.2 5.1 – 5.1.1	November 4, 2014 March 2, 2015
Android Marshmallow	Macadamia Nut Cookie	6.0 – 6.0.1	October 2, 2015
Android Nougat	New York Cheesecake	7.0 7.1 – 7.1.2	August 22, 2016 October 4, 2016
Android Oreo	Oatmeal Cookie	8.0 8.1	August 21, 2017 December 5, 2017
Android Pie	Pistachio Ice Cream	9	August 6, 2018
Android 10	Quince Tart	10	September 3, 2019
Android 11	Red Velvet Cake	11	September 8, 2020
Android 12	Snow Cone	12	October 4, 2021
Android 12L	Snow Cone v2	12.1	March 7, 2022
Android 13	Tiramisu	13	Q3 2022

offers the information and source code needed to create custom variants of the Android OS, port devices and accessories to the Android platform, and ensure devices meet the compatibility requirements that keep the Android ecosystem a healthy and stable environment for millions of users.

As an open-source project, Android's goal is to avoid any central point of failure in which one industry player can restrict or control the innovations of any other player. To that end, Android is a full, production-quality operating system for consumer products, complete with customisable source code that can be ported to nearly any device and public documentation available to everyone.

At webpage `https:\\cs.android.com`, source codes for Android, the AndroidX Library or the Android Studio development environment are available.

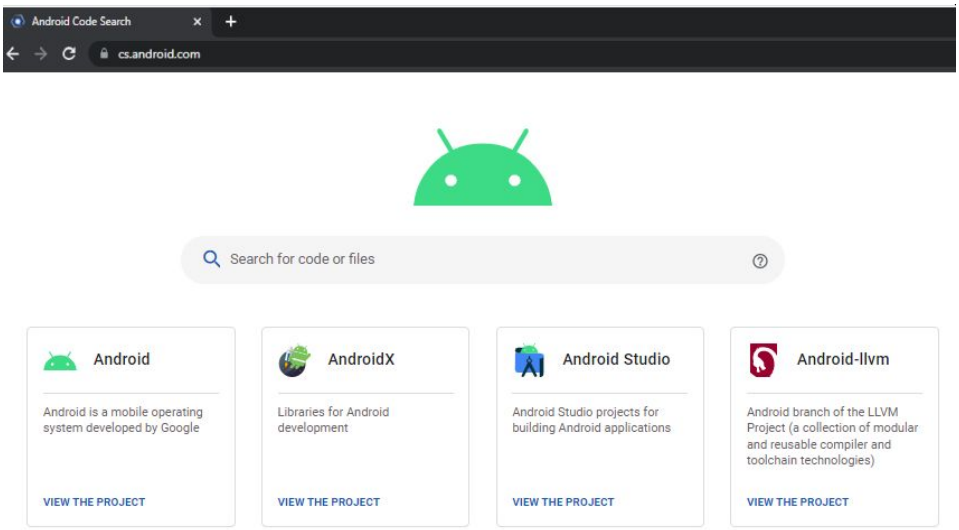


Figure 1.3: Android Code Search [12]

Development tools

Effective software development requires usage tools for developing and testing applications. It is also essential to choose the right language and library or framework. The course described in this book focuses on the basics; hence the choice of libraries does not go beyond the official proposals.

2.1 Android Studio

The freeware Android Studio is used for programming. The development environment is available for multiple platforms, including Windows, Mac, Linux, and Chrome OS [13].

Android Studio is the official Integrated Development Environment (IDE) for Android app development, based on IntelliJ IDEA. On top of IntelliJ's powerful code editor and developer tools, Android Studio offers even more features that enhance your productivity when building Android apps, such as [14]:

- A flexible Gradle-based build system
- A fast and feature-rich emulator
- A unified environment where you can develop for all Android devices
- Apply Changes to push code and resource changes to your running app without restarting your app
- Code templates and GitHub integration to help you build common app features and import sample code
- Extensive testing tools and frameworks
- Lint tools to catch performance, usability, version compatibility, and other problems
- C++ and NDK support
- Built-in support for Google Cloud Platform, making it easy to integrate Google Cloud Messaging and App Engine

Android Studio is released under the open-source Apache License 2.0. However, it also includes some proprietary code Figure no. 2.1.

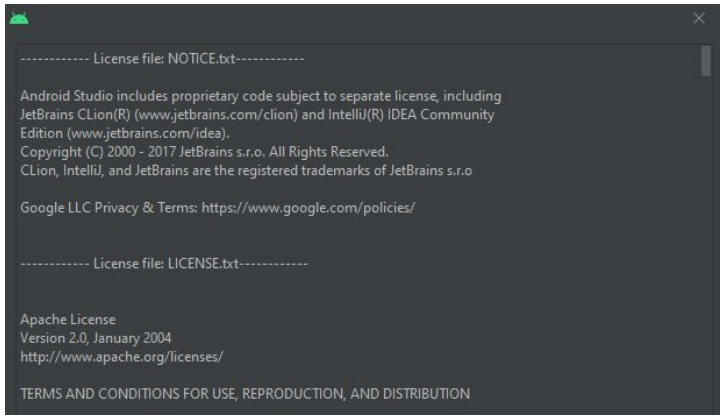


Figure 2.1: Android Studio licence

2.2 Programming language

The primary language currently used for Android app development is Kotlin, which has replaced the JAVA language. Kotlin is a relatively new language, having only reached 1.0 status in 2016. At the 2019 Google I|O conference, Google announced that the Android SDK would be developed as "Kotlin first" in the future. Although programming in Java is still possible, Google will focus on Kotlin, particularly regarding documentation, tutorials, or educational material.

Kotlin adopts newer approaches, and the resulting language can be much more concise so that more work can be done with fewer lines of code. At the same time, while helping to minimise errors in the code.

2.2.1 Kotlin

Kotlin is a cross-platform, statically typed general-purpose programming language. Kotlin is designed to be fully interoperable with Java, and the Kotlin standard library for the JVM depends on the Java class library.

Kotlin is a modern language that is easier to use and more readable than Java. The ease and simplicity of the language make it more reliable for applications. Features of Kotlin that make it considered easier:

- Optional semicolons,
- Create objects without *new*,
- Skip variable type,
- Optional *void* in functions,
- Classes can be much shorter than in JAVA,
- Static typing,
- Checking *null*, need to explicitly specify the object as *nullable*,

- Handling of named and optional arguments,
- Lambda functions,
- Kotlin does not have checked exceptions.

The language developers at `https://kotlinlang.org/#why-kotlin` provide several examples demonstrating the advantages of the language. Two of them will be quoted here [9]:

- **Concise:**

```
data class Employee(
    val name: String,
    val email: String,
    val company: String
) // + automatically generated equals(), hashCode(), toString(), and copy()

object MyCompany{                           // A singleton
    const val name: String = "MyCompany"
}

fun main(){
    val employee = Employee("Alice",        // No `new` keyword
        "alice@mycompany.com", MyCompany.name)
    println(employee)
}
```

- **Safe:**

```
fun reply(condition: Boolean): String? =          // Nullability is part of 'Kotlins type system
    if (condition) "I'm fine" else null

fun error(): Nothing =                            // Always throw an exception
    throw IllegalStateException("Shouldn't be here")

fun main(){
    val condition = true                         // Try replacing `true` with `false` and run the sample!
    val message = reply(condition)               // The result is nullable
    // println(message.uppercase())              // This line doesn't compile
    println(message?.replace("fine", "okay"))    // Access a nullable value in a safe manner
    if (message != null){                       // If you check that the type is right,
        println(message.uppercase())             // the compiler will smart-cast it for you
    }

    val nonNull: String =                             // If the null-case throws an error,
    reply(condition = true) ?: error()               // Kotlin can infer that the result is non-null
    println(nonNull)
}
```

- **Expressive**
- **Interoperable**
- **Multiplatform**

More information about the Kotlin language can be found in the documentation [8], and it is worth reading the book [10].

2.3 Integrated Development Environment

The Android Studio environment provides many tools which let us create applications faster. Among the many Android Studio tools, we can distinguish:

- Intelligent code editor

- Visual layout editor
- Device Emulator
- Instant App Run
- Testing tools and frameworks
- Wireless debugging
- App Inspection – Database Inspector, Network Inspector, Backgroud Task Inspector
- Code Analyze

2.3.1 Testing applications in Android Studio

The Android Studio environment and the tools built into the SDK provide many tools to support application testing. The most important is the ability to run/debug applications on real devices and emulators. You launch and control the application directly from the environment.

If you want to install the application on a physical device directly from Android Studio, you must first enable the developer options on the device, which are hidden. This is unlocked by tapping seven times on the "Build Number", which can be found in "**Settings → About Phone → Build Number**". In the next step, using the USB cable, we can already install, run and debug the application. In recent releases of Android Studio, the possibility to run the application using a WiFi wireless connection has been added.

Another possibility to test the created application is to use the Android emulator. An image of this system is distributed together with the SDK. In addition to running and debugging, the tool allows for to change of sensor values, emulating positioning, emulating camera images and changing network parameters. This enables testing of most device use cases. More information about the emulator can be found at `https://developer.android.com/studio/run/emulator`.

In addition to running, it is possible to use a window for reading logs called Logcat. It displays messages in real-time and keeps a history so that older messages can be viewed. We can filter messages according to various criteria. If an exception has not been handled in the application, the *Logcat* window displays the error information `https://developer.android.com/studio/debug/am-logcat`.

Android Studio has built-in tools that are used to build and validate the correctness of the built layout. These are: Layout Inspector and Layout Validation. The first tool is used to validate the user interface, allowing the entire hierarchy to be displayed, which makes it easier to work with complex layouts. Layout Inspector is available via "**Tools → Layout Inspector**" for devices with APIs above 29, it is possible to run a Live Layout Inspector mode, which allows the user interface to be viewed while working on an emulator or physical device. The Layout Validation tool allows the layout to be reviewed simultaneously on different devices and with different screen settings(e.g. font size, device language).

Tools are also built into the environment to streamline the creation of automated application tests. On page `https://developer.android.com/training/testing`,

good practices related to application testing are included. The available tools are also presented.

The tools mentioned above are the basic ones used in application development. It is worth reading a complete description of application testing tools `https://developer.android.com/studio/debug`.

Application Fundamentals

The Android app is distributed via a single apk file. The ordinary user installs the application via the Google Play market. It is also possible to use alternative shops, including manufacturers' shops (e.g. Samsung Galaxy Store). The Android Studio environment allows applications to run on both a virtual device (emulator) and a physical device. Once an app is installed, the system launches it, with each Android app running in a security sandbox. This helps to protect the app and the user. The following rules apply:

- Security is based on the permission rules as they are in Linux,
- Each application is a different user,
- Each process is a separate virtual machine,
- An application is running when one of its components needs to be run.
- Access to some system resources (e.g. location) requires permission from the user

Developing applications in a mobile system requires knowledge of the available components, permissions or system limitations. As the system has developed, the recommendations for developing applications or the capabilities of the available libraries have changed, and restrictions have emerged to protect the user's privacy or conserve the device's energy.

3.1 Components

When creating a programme, we use four basic types of components:

- Activities
- Services
- Broadcast receivers
- Content providers

Activity – It is a key application component. An Activity represents a single window of the application. It allows interaction with the user. There can be multiple Activities in each application, each with its life cycle.

Service – Is a component that allows background operations, e.g. performing long-running computational operations or waiting for calls from others (external or internal services). Services do not provide a user interface. It is possible to associate an activity with a service and perform an interaction.

Broadcast receivers – The component allows the system to deliver messages between application components and different applications or the system. System messages such as "Battery low" are particularly useful. The component does not have a user interface but can send notifications so the user can take action.

Content providers – A mechanism that allows sharing data with other applications using a uniform interface in the form of URI addresses. Also, the system provides its data using the Content providers mechanism so that we can use a database of photos or a database of phone numbers.

Each component has its life cycle, and these will be presented later.

Launching components (except for Content providers) is possible through intents (Intent class). Intents facilitate communication between components in several ways, and there are three basic use cases:

- Starting an activity
- Starting a service
- Delivering a broadcast

Intents can be defined as explicit or implicit. **Explicit**, i.e. the full class name of the component is defined, or **Implicit**, i.e. the name of the component is not specified, but the action to be performed is declared. An action specifies an activity that can be performed, e.g. display, edit or send.

```
// Build the intent.
val location = Uri.parse("geo:0,0?q=1600+Amphitheatre+Parkway,+Mountain+View,+California")
val mapIntent = Intent(Intent.ACTION_VIEW, location)

// Try to invoke the intent.
try{
    startActivity(mapIntent)
} catch (e: ActivityNotFoundException){
    // Define what your app should do if no activity can handle the intent.
}
```

Listing 3.1: Example of implicit intent – View a map

We can create any number of components and use them repeatedly depending on the application's needs. Still, it is necessary to declare them in the application manifest, i.e. the file **AndroidManifest.xml**.

3.2 Android Manifest

The manifest file is a mandatory file that must be included in the project. Its name is "AndroidManifest.xml", located in the application's root directory. The file defines the relevant information about the application for the Android build tools, the Android operating system and the Google Play shop. In particular, we need to include information about the application name, components, hardware requirements, and permissions.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <uses-permission android:name="android.permission. ..."/>
    <uses-future android:name="android.hardware. ..."/>

     <application android:icon="@drawable/app_icon.png" ... >
         <activity android:name="com.example.project.ExampleActivity"
                   android:label="@string/example_label" ... >
         </activity>
         <service>
         </service>
         <receiver>
         </receiver>
         <provider>
         </provider>
         ...
     </application>
</manifest>
```

Listing 3.2: Skeleton of the AndroidManifest.xml file

In particular, it must contain information about all activities, services, broadcast receivers, and content providers. A component must have at least a class name defined, but we can also add an intent filter or hardware requirements. The skeleton of the manifest is shown in the listing 3.2. It contains only basic properties, detailed documentation is available at `https://developer.android.com/guide/topics/manifest/manifest-intro` [17].

```xml

<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1"
    android:versionName="1.0">

    <!-- Beware that these values are overridden by the build.gradle file -->
    <uses-sdk android:minSdkVersion="15" android:targetSdkVersion="26" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-feature android:name="android.hardware.sensor.compass"
                  android:required="true" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">

        <!-- This name is resolved to com.example.myapp.MainActivity
             based upon the namespace property in the 'build.gradle' file -->
        <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=".DisplayPositionActivity"
            android:parentActivityName=".MainActivity" />
    </application>
</manifest>
```

Listing 3.3: Manifest file of the sample application

From the manifest file of the example Listing no. 3.3, it can be read that the application consists of two activities (**MainActivity** and **DisplayMessageActivity**), using an intent filter, it was specified that the **MainActivity** activity would be started as soon as the application starts. Permission to use the exact position is required for the application to run, and the device must have a compass sensor.

3.3 Lifecycle

The LifeCycle is a class/interface that stores information about the state of components, including Activity/Fragment and allows other objects to observe this state by tracking it. The LifeCycle component deals with the lifecycle events of an Android component, such as Activity or Fragment, and contains three main classes:

- Lifecycle
- Lifecycle Owner
- Lifecycle Observer

3.3.1 Activity lifecycle

Activities are key components in an application and are responsible for interacting with the user. Knowing the lifecycle of this component is crucial to building the application correctly. Each activity receives a window in which it draws its user interface. The window usually fills the whole screen; sometimes, it can be smaller. An application may consist of multiple activities that are loosely connected. Activity in its life cycle can take on different states:

- **Initialized** – Activity instance is created, and its properties are initialised.
- **Created** – Activity is now completely initialised and ready to configure its UI.
- **Started** – Activity is visible to the user.
- **Resumed** – Activity is visible to the user and has focus. In this state, the user is likely interacting with the activity.
- **Destroyed** – Activity is destroyed, and the OS can reclaim its memory.

```
class MainActivity : AppCompatActivity(){

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

   override fun  onStart(){
      super.onStart();
   }

   override fun onRestart(){
      super.onRestart();
   }

   override fun onPause(){
      super.onPause();
   }

   override fun onResume(){
      super.onResume();
   }

   override fun onStop(){
      super.onStop();
   }

   override fun onDestroy(){
      super.onDestroy();
   }
}
```

Listing 3.4: Activity lifecycle calback

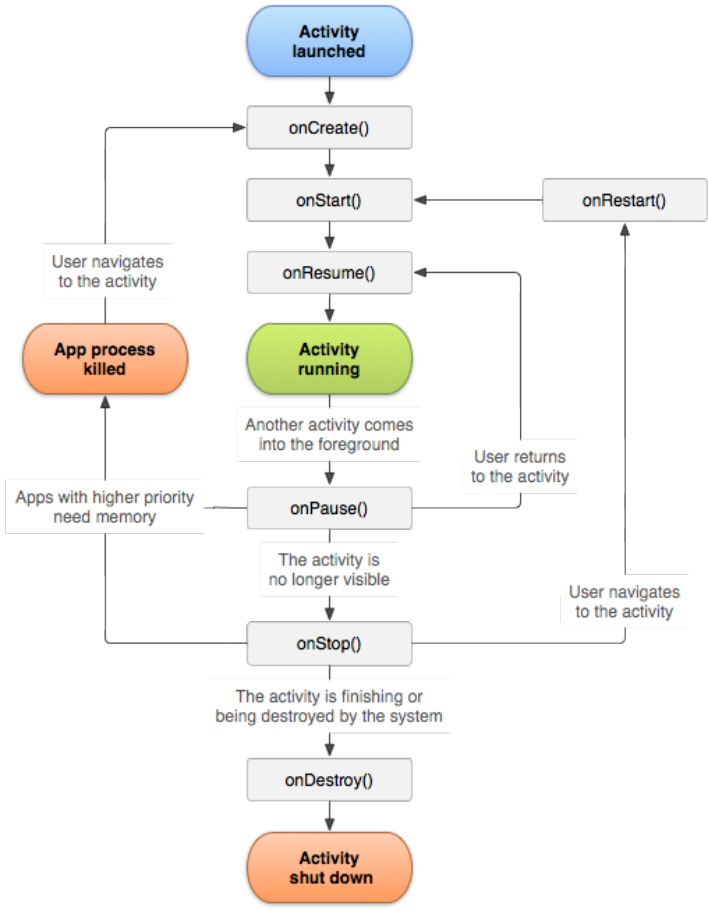


Figure 3.1: Activity Lifeycle [16]

Figure 3.1 shows all the methods from the activity lifecycle. Below is a description of the individual callbacks methods:

- **onCreate()** – The activity enters state **Initialized**. This is where logic is performed that should only occur once during the activity's lifecycle. This may include setting the content view, binding the activity to a ViewModel, initialising some class scope variables, etc.
- **onStart()** – The activity enters state **Started**. This call makes the activity visible to the user as the application prepares for the activity to come to the foreground and become interactive.
- **onResume()** – The activity enters state **Resumed**. The user can now interact with the activity. This is where you can enable all the functions that must be activated when the component is visible and in the foreground.

- **onPause()** – The activity transitions to state **Paused**. This call indicates that the activity is no longer in the foreground, although it may still be visible if the user uses multi-window mode, for example. During this time, operations that should not be continued or should be continued in moderation should be paused or adjusted. The activity remains in this state until it resumes operations – for example, opening or closing the bottom activity sheet – or until it becomes completely invisible to the user – for example when opening another activity.
- **onStop()** – The activity enters state **Stopped**. The activity is no longer visible to the user. At this point, you should release or adjust resources that are not needed when the activity is not visible to the user. You should also use this opportunity to perform shutdown operations on relatively CPU-intensive tasks, such as database operations.
- **onDestroy()** – The activity transitions to state **Destroyed**. At this point, the activity stops. This may be because the user has closed the activity or a method *finish()* has been called. function *finish()* is called on the activity.

3.3.2 Fragments

Activities are quite heavy on the system, making managing data and multiple lifecycles cumbersome. As the system developed, the ability to create fragments was added. They were initially provided to make developing applications for different screen sizes easier. This mainly applied to tablets, where the look of an app could be assembled from several fragments. Nowadays, Google's recommended practice is creating an activity, making changes to functionality, and including the user interface using fragments. Fragments have their lifecycle, are always nested within an activity, and their lifecycle is linked to the hosting activity.

Currently, fragment support is implemented as part of package **JetPack – AndroidX Fragments**.

Each fragment instance has its life cycle linked to the activity that started it. When the activity is destroyed, so are all the fragments, when the activity is suspended, so are the fragments. The user interacts with the application while working with it with individual fragments that change states in their life cycle. **LifecycleObserver** allows the programmer to detect when a fragment is active. This allows specific actions to be performed. For example, the application can display a message **Snackbar** or **Toast**. Similar to the activity, the fragment interface can be built using XML. Fragments can be in the following states:

- INITIALIZED
- CREATED
- STARTED
- RESUMED
- DESTROYED

3.3.3 Methods of the Android Fragment

Callback methods are used to manage the life cycle of a fragment. These functions include well-known activity methods such as *onCreate, onStart, onResume, onPause, onStop, onDestroy*. A few new ones also arrive.

- **onAttach()** – is called when a fragment is attached to an activity.
- **onCreate()** – is called to perform the initialisation of a fragment.
- **onCreateView()** is called by Android when the fragment should output a view.
- **onViewCreated()** – is called after onCreateView() – called when the fragment's root view is created. Any view configuration should take place here.
- **onActivityCreated()** – is called when the host activity has completed the onCreate() method.
- **onStart()** – is called when the fragment is ready to be displayed on the screen.
- **onResume()** – in this method, we trigger the action of any listeners, e.g. reading locations, sensors, etc.
- **onPause()** – in this method, we release the resources started in onResume(). We save any data.
- **onDestroyView()** – is called when the fragment view is destroyed.
- **onDestroy()** – is called when the fragment is no longer in use.
- **onDetach()** – is called when a fragment is no longer connected to an activity.

Fragments are closely linked to View, which is shown in figure no. 3.2.

3.3.4 Creating fragments

The first step is to add the relevant libraries to the **build.gradle** file 3.5.

```
dependencies{
      // Kotlin
     implementation("androidx.fragment:fragment-ktx:fragment_version")
}
```

Listing 3.5: Defining dependencies

Then create a class that inherits from the Fragment class from the AndroidX package 3.6. Specialised base classes, such as **DialogFragment** or **PreferenceFragmentCompat**, have also been created, allowing for faster dialogue creation or window for application settings.

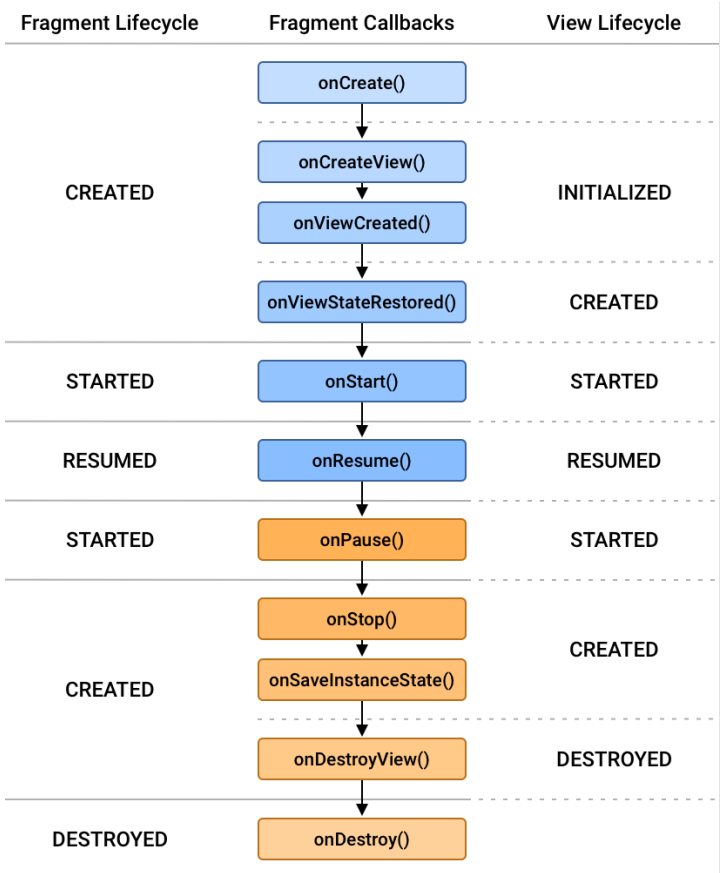


Figure 3.2: Fragment Lifeycle [18]

```
...
import androidx.fragment.app.Fragment
...
class ExampleFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val binding = DataBindingUtil.inflate<FragmentExampleBinding>(inflater,
                R.layout.example_fragment, container, false)
        return binding.root
    }
}
```

Listing 3.6: Example Fragment

Inside the class, the lifecycle callback methods should be used, including *onCreateView()*, where we can assign the fragment's layout and then bind UI objects and perform operations on them, as shown in the example code.

The next step is to add the fragment to the activity. To do this, *FragmentContainerView* (Listing 3.7) can be added to the activity's layout file, which defines where the fragment should be placed in the activity's view hierarchy, *android:name* indicates the class responsible for the fragment.

Another way is to define only the placeholder without specifying the class name (Listing 3.7) and then programmatically define which fragment will be displayed (Listing 3.8). We make the changes using instances of the class **FragmentManager**. This approach allows us to change the UI (fragment) during user interaction.

```
<!-- res/layout/example_activity.xml -->
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Listing 3.7: Define place for fragment (any)

```
class ExampleActivity : AppCompatActivity(R.layout.example_activity){
    override fun onCreate(savedInstanceState: Bundle?){
        super.onCreate(savedInstanceState)
        if (savedInstanceState == null){
            fragmentManager.commit{
                setReorderingAllowed(true)
                add<ExampleFragment>(R.id.fragment_container_view)
            }
        }
    }
}
```

Listing 3.8: Add a fragment programmatically

Managing fragments and their lifecycles requires much work. Android Jetpack introduced **Navigation Component**, which simplifies interaction in the application and allows changes to fragments to be managed.

3.4 Navigation Component

The Navigation Component is a set of libraries and tools to simplify the implementation of navigation. It consists of three parts:

- **Navigation Graph** – an XML file containing all the navigation information in the application. All the paths or steps the user can take in the application are defined.
- **Nav Host** – an empty container in which destinations are swapped as the user moves through the application.
- **NavController** – the object that manages the application's navigation within *NavHost*.

The Navigation component provides the following:

- Fragment support,
- Default support for "Up" and "Back" actions,
- Allows the use of standard animations and transitions between actions.
- Implementation of navigation patterns (such as navigation drawers or bottom navigation),

- Support for Gradle's plugin, Safe Args, which improves the passing of data between fragments,
- Support for ViewModel – a ViewModel can be attached to a navigation graph to share UI-related data between graph destinations.

These advantages, taken together with the tools provided by Android Studio (Navigation Editor [19]), mean that this component simplifies managed fragments in the application. Using this component boils down to the following steps:

1. Set up your environment
2. Create a navigation graph
3. Definition of NavHost in Activity
4. Assignment of actions

Set up your environment Adding dependencies to the project (build.gradle file) Listing no. 3.9

```
dependencies {
    implementation("androidx.navigation:navigation-fragment-ktx:nav_version")
    implementation("androidx.navigation:navigation-ui-ktx:nav_version")

    // Feature module Support
    implementation("androidx.navigation:navigation-dynamic-features-fragment:nav_version")
}
```

Listing 3.9: Adding dependencies

Create a navigation grap Navigation graphs are XML files that encapsulate the definition of potential user paths that may arise during interaction with the application. A tool is provided in Android Studio that visually supports the creation of navigation descriptions. Figure 3.3 shows an example of navigation. Four different actions have been defined for three different fragments. The corresponding XML code for this graph is in Listing 3.10.

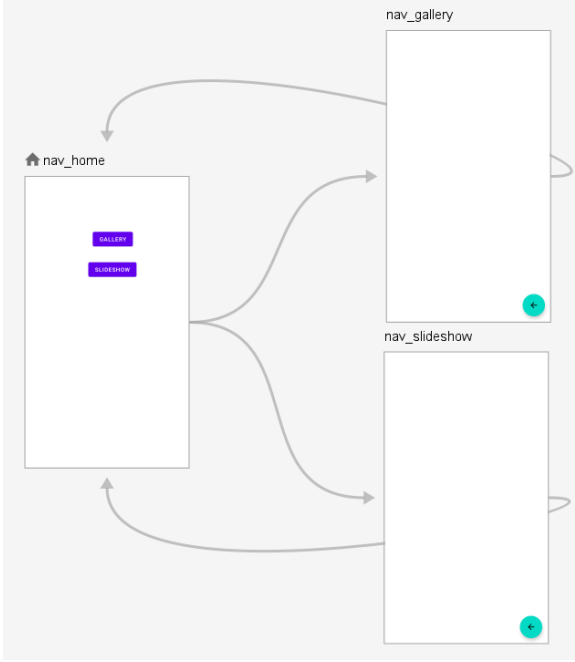


Figure 3.3: Navigation graph – tools view

```xml
<?xml version="1.0" encoding="utf-8"?>
<navigation 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:id="@+id/mobile_navigation"
    app:startDestination="@+id/nav_home">

    <fragment
        android:id="@+id/nav_home"
        android:name="edu.zut.erasmus_plus.examplenavigation.ui.home.HomeFragment"
        android:label="@string/menu_home"
        tools:layout="@layout/fragment_home" >
        <action
            android:id="@+id/action_nav_home_to_nav_gallery"
            app:destination="@id/nav_gallery" />
        <action
            android:id="@+id/action_nav_home_to_nav_slideshow"
            app:destination="@id/nav_slideshow" />
    </fragment>

    <fragment
        android:id="@+id/nav_gallery"
        android:name="edu.zut.erasmus_plus.examplenavigation.ui.gallery.GalleryFragment"
        android:label="@string/menu_gallery"
        tools:layout="@layout/fragment_gallery" >
        <action
            android:id="@+id/action_nav_gallery_to_nav_home"
            app:destination="@id/nav_home" />
    </fragment>

    <fragment
        android:id="@+id/nav_slideshow"
        android:name="edu.zut.erasmus_plus.examplenavigation.ui.slideshow.SlideshowFragment"
        android:label="@string/menu_slideshow"
        tools:layout="@layout/fragment_slideshow" >
        <action
            android:id="@+id/action_nav_slideshow_to_nav_home"
            app:destination="@id/nav_home" />
    </fragment>
</navigation>
```

Listing 3.10: Navigation graph – XML

We define the initial interface using the *app:startDestination* property (line no. 6). Then, we define the individual fragments and the actions that can be performed. In our case, two actions are defined for fragment *edu.zut.erasmus_plus.examplenavigation.ui.home.HomeFragment*. In the code, these actions are assigned to the corresponding buttons.

3.4.1 Definition of NavHost in activity

Navigation host is an empty container responsible for displaying individual fragments. It can cover the whole screen or be a specific part.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:app="http://schemas.android.com/apk/res-auto"
4      xmlns:tools="http://schemas.android.com/tools"
5      android:layout_width="match_parent"
6      android:layout_height="match_parent"
7      app:layout_behavior="@string/appbar_scrolling_view_behavior"
8      tools:showIn="@layout/app_bar_main">
9
10     <androidx.fragment.app.FragmentContainerView
11         android:id="@+id/nav_host_fragment_content_main"
12         android:name="androidx.navigation.fragment.NavHostFragment"
13         android:layout_width="match_parent"
14         android:layout_height="match_parent"
15         app:defaultNavHost="true"
16         app:layout_constraintLeft_toLeftOf="parent"
17         app:layout_constraintRight_toRightOf="parent"
18         app:layout_constraintTop_toTopOf="parent"
19         app:navGraph="@navigation/mobile_navigation" />
20 </androidx.constraintlayout.widget.ConstraintLayout>
```

Listing 3.11: Navigation navhost

It is worth noting in the example Listing no. 3.11:

- **android:name** – attribute contains the class name of your NavHost implementation,
- **app:navGraph** – attribute associates the NavHostFragment with a navigation graph,
- **app:defaultNavHost="true"** attribute ensures that your NavHostFragment intercepts the system Back button. Note that only one NavHost can be the default.

Assignment of actions Actions in the program are often assigned to buttons by adding an event handler *onClick*.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  override fun onClick(view: View){
3      view.findNavController().navigate(@+id/action_nav_home_to_nav_gallery)
4  }
```

Listing 3.12: Calling a navigation action from method *onClick()*.

The above example, Listing no. 3.12 proceeds to the given fragment without passing arguments. Argument passing is possible by using **Safe Args Gradle plugin**.

Before the introduction of Safe Args, passing data when passing to another screen required using an object *Bundle*. The first Activity, the sender, created an instance *Bundle* and filled it with data. The second Activity, the receiver, retrieved this data later. This manual approach to sending and unpacking data is unreliable because the sender and receiver must agree on the following:

- Keys,
- Default values for each key,
- The type of data corresponds to the keys.

Nor can force the recipient or the sender to provide all the required data. Furthermore, when unpacking data, type security is not guaranteed on the recipient's side.

Android has introduced Safe Args to address these issues. **Safe Args** is a plug-in to **Gradle** that generates code to add data to the *Bundle* packet and access it in a simple and type-safe manner.

Using **Safe args** requires changes to the build.gradle files. To do this, we need to add Listing plugin no. 3.13 in build.gradle.

```
<!-- top-level build.gradle -->
dependencies{
...
  classpath "androidx.navigation:navigation-safe-args-gradle-plugin:navigation_version"
}

<!-- app-level build.gradle -->
<?xml version="1.0" encoding="utf-8"?>
plugins{
    id("androidx.navigation.safeargs")
   \\ or when generate Kotlin code suitable for Kotlin-only modules
   id("androidx.navigation.safeargs.kotlin")

apply plugin: "androidx.navigation.safeargs"
}
```

Listing 3.13: Set up Safe Args

When Safe Args is enabled, this plugin generates code containing classes and methods for each defined action. For each action, Safe Args also generates a class for each destination, i.e. the destination from which the action originates. The use of arguments requires that argument information is added in the navigation graph Listing no. 3.14. In this example, we expect information about the value of argument *startImage* in fragment *GalleryFragment*.

```

<!-- res/navigation/mobile_navigation.xml -->
...
    <fragment
        android:id="@+id/nav_gallery"
        android:name="edu.zut.erasmus_plus.examplenavigation.ui.gallery.GalleryFragment"
        android:label="@string/menu_gallery"
        tools:layout="@layout/fragment_gallery" >
        <argument
            android:name="startImage"
            app:argType="integer"
            android:defaultValue="1" />
        <action
            android:id="@+id/action_nav_gallery_to_nav_home"
            app:destination="@id/nav_home"/>
    </fragment>
...
```

Listing 3.14: Navigation Arguments

The action involved in passing the argument is shown in Listing no. 3.15.

```
override fun onClick(view: View){
   val actionWithArgs = HomeFragmentDirections.actionNavHomeToNavGallery().setStartImage(2)

    view.findNavController().navigate(actionWithArgs)
}
```

Listing 3.15: Navigation action arguments

It is worth noting in the example Listing no. 3.15:

- HomeFragmentDirections - **Safe Args** automatically generates a class with a name corresponding to the fragment class name and the word **Directions**,
- actionNavHomeToNavGallery – **Safe Args** automatically generates an action object with a name without underscore characters,
- setStartImage – **Safe Args** automatically generates a method to set an argument value.

A mechanism created this way is simple to implement and avoids errors by reading values in the target fragment.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?){
    ...
    val args: GalleryFragmentArgs by navArgs()
    val startImage = args.startImage
    ...
}
```

Listing 3.16: Calling the navigation action from the method *onClick()*

In Listing no. 3.16, we use the class *GalleryFragmentArgs* automatically created by mechanism SafeArgs, from where we read the passed value. In this way, we can pass not only simple types but also object types.

For more on parameter passing, see documentation [20].

3.5 Services

A service is a component that allows long-lasting operations such as calculations, waiting for events or network communication. It can run for long periods, even if the user is dealing with another application. The main components of android can bind to a service to interact with it. Services can also perform inter-process communication. We can distinguish between three types of services:

- Foreground – perform tasks that are noticeable to the user, e.g. playing music. They need to display notifications even when not interacting directly with the user. The notification is displayed until the service is running.
- Background – perform background tasks that do not notify the user that they are running.
- Bound – perform tasks linked to other running components. The lifecycle of a service is closely related to the lifecycle of other components linked to the service. Many components can be linked at one time. When the last one disconnects then, the service is destroyed.

Another division is services of type started – unbounded service and type bounded. **Started**, terminate after a task has been performed, while **bounded** run until the last component associated with it. It is possible to create a service that is both of type **started** and **bound**. The difference in type lies in how the service is started and, thus, the implementation of the callback methods. Figure no. 3.4 shows the order in which the methods are called and the service states.

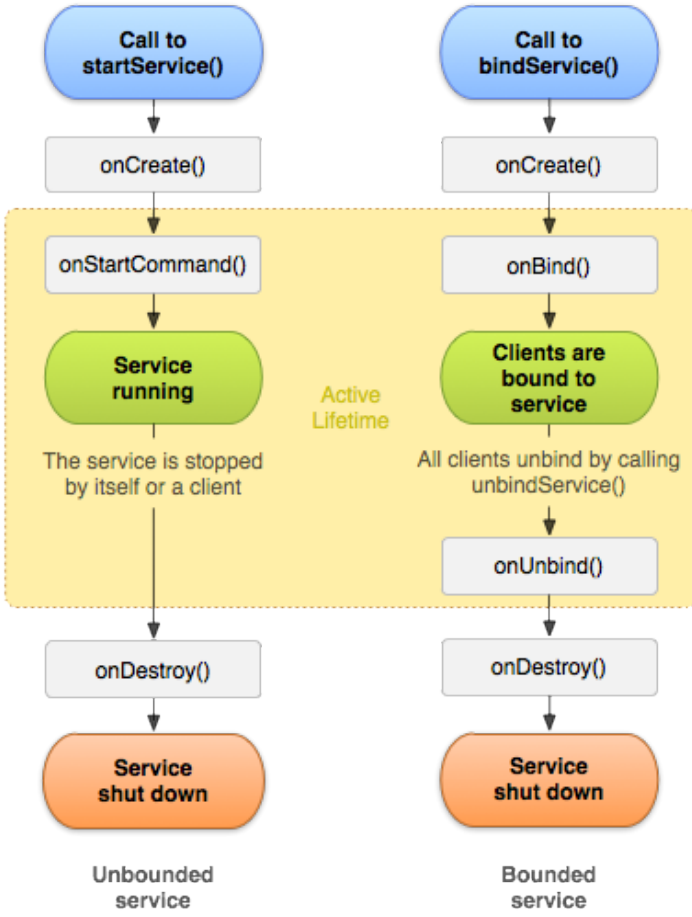


Figure 3.4: Service lifecycle [21]

Services *started* are started when another component calls a service using method *startService()*. The system then calls methods *onCreate*, *onStartCommand()*, when the task is finished, a method *onDestroy()* is called. Services of type *bounded* are started when another component calls the bindService() method. The system then calls methods *onCreate*, *onBind()*. When the last component disconnects, the onUnbind() method is called, and then method *onDestroy()* is called. It is noteworthy that when one component starts a service of type *started*, the system calls method *onBind()* when the next component connects.

3.5.1 Creation of services

For the service to be run from the application, it must be declared in the AndroidManifest.xml file 3.17. Only the class name is required.

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
</manifest>
```

Listing 3.17: Example of service declaration

We then declare the class of our service by extending class *Service*. The skeleton of the class is shown in Listing no. 3.18. It is worth noting that, unlike in *Activity*, we do not need to refer to the base class in the methods.

```
class ExampleService : Service(){
    private var startMode: Int = 0             // indicates how to behave if the service is killed
    private var binder: IBinder? = null        // interface for clients that bind
    private var allowRebind: Boolean = false   // indicates whether onRebind should be used

    override fun onCreate(){
        // The service is being created
    }

    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int{
        // The service is starting, due to a call to startService()
        return startMode
    }

    override fun onBind(intent: Intent): IBinder?{
        // A client is binding to the service with bindService()
        return binder
    }

    override fun onUnbind(intent: Intent): Boolean{
        // All clients have unbound with unbindService()
        return allowRebind
    }

    override fun onRebind(intent: Intent){
        // A client is binding to the service with bindService(),
        // after onUnbind() has already been called
    }

    override fun onDestroy(){
        // The service is no longer used and is being destroyed
    }
}
```

Listing 3.18: Skeleton service [22]

We use *Intent* to launch services. We start the intention by calling the appropriate method depending on the type. In the case of a service of type **Started**, this is *startService()*.

```
Intent(this, ExampleService::class.java).also{ intent ->
    startService(intent)
}
```

Listing 3.19: Launching the service

The topics of the services are quite complex, especially services of the type *Bounded*. More information can be found in documentation [23], and [21].

A separate issue is the communication of the results of the services. A recommended method is to use the component **Broadcast Receivers**.

3.6 Broadcast Receivers

Bradcast Receivers is a component used to transfer messages between application components. This way, the application can inform about the data download and inform where

the data is stored. The component offers great possibilities when receiving system messages. The application can receive information about system start-up, low battery or, for example, about connecting a device for charging. Once registered to receive a particular message, the Android system mechanism automatically ensures that the message is received. The app's messages are distributed the same way as system messages.

3.6.1 Receiving broadcasts

Receiving messages is possible in two ways. The first is by registering the receiver in the manifest file. The second is to register the receiver while the application is running. This second way means that messages will only be received when the application runs. An example of registration in the AndroidManifes.xml file is shown in Listing no. 3.20. The parameter *android:exported="true"* specifies that it is possible to receive messages from other applications. We are registering for the standard defined actions sent by the system; therefore, the attribute takes the value "*true*".

```
<receiver android:name=".ExampleReceiver"  android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
        <action android:name="android.intent.action.INPUT_METHOD_CHANGED" />
    </intent-filter>
</receiver>
```

Listing 3.20: Declaration of receiver in AndroidManifest.xml

Once registered, we need to create a class responsible for receiving the message. It must inherit from class *BradcastReceiver* 3.21 and has one callback method **onReceive**. In the body of this method, it implements the code to be executed when the action is received. It is worth noting that the onReceive method receives all actions declared for this class.

```
class ExampleReceiver : BroadcastReceiver(){

    override fun onReceive(context: Context, intent: Intent){
    ...

   }
}
```

Listing 3.21: Example Broadcast Receiver class

The implementation of the second method involves a method called *registerReceiver()*. The message handler class instance object and the intention filter responsible for the action are given as arguments 3.22.

```
...
val exampleReceiver: BroadcastReceiver = ExampleReceiver()
...

val filter = IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION).apply{
    addAction(Intent.ACTION_AIRPLANE_MODE_CHANGED)
}
registerReceiver(br, filter)
}
```

Listing 3.22: Context-registered receivers

3.6.2 Sending broadcasts

Sending a message is performed with the help of an object *Intent*, which is passed to one of several possible broadcast sending methods.3.23.

```
//Sending to own action
Intent().also{ intent ->
    intent.setAction("edu.zut.erasmus_plus.example.MY_ACTION")
    intent.putExtra("data", "Send with data")
    sendBroadcast(intent)
   }
//Sending to action  with permissions

Intent().also{ intent ->
    intent.setAction("edu.zut.erasmus_plus.example.my_action")
    intent.putExtra("data", "Send with data")
    sendBroadcast(intent,Manifest.permission.SEND_SMS)
   }
```

Listing 3.23: Sending bradcast – example

If we define permissions in the method *sendBroadcast*, only applications that have the given permission declared can receive this message 3.24.

```
<receiver android:name=".ExampleReceiver"  android:exported="true">
   android:permission="android.permission.SEND_SMS">
    <intent-filter>
        <action android:name="edu.zut.erasmus_plus.example.MY_ACTION"/>
    </intent-filter>
</receiver>
```

Listing 3.24: Declaration of receiver with permission in AndroidManifest.xml

3.7 Content Provider

The purpose of the *Content Provider* component is to manage access to data in an application that is structured. The component is part of an Android application, which often provides its own UI for working with data. It is generally used to make data available to other applications, however, using the *Content Provider* component exclusively inside the application also offers many advantages. The main advantages of using *Content Provider*:

- creating a standard interface for accessing data,
- securing the data against both access and data corruption
- providing a search mechanism using suggestions

The *Content Provider* provides data in the form of one or more tables, similar to a relational database. The component coordinates access to the application's data storage layer for several different APIs and components, as shown in figure no. 3.5, these include:

- Sharing access to your application's data with other applications,
- Synchronising application data with the server using *AbstractThreadedSyncAdapter*,
- Downloading data to the user interface using *CursorLoader*.

Access to the data is performed by defining a URI (Uniform Resource Identifier) address, which is described by document RFC2396 [29].

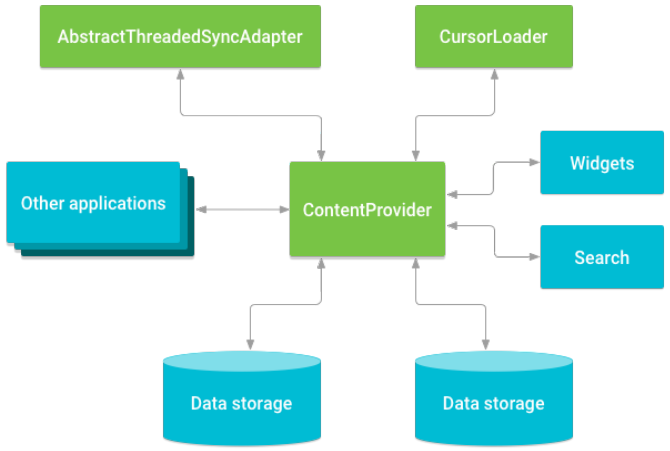


Figure 3.5: Relationship between content provider and other components [28]

3.7.1 Creation of content providers

Creating the *content providers* requires that the data be prepared accordingly. They should be in a structured form so that it is possible to store them in the form of tables and thus store them in a database. This requires the creation of a database, and this is one of the steps in creating *content providers*. It is also possible to provide file data stored in the application's private memory. The next step is implementing a custom class extending the *Content Providers* class. The implementation requires the creation of six callback methods.

- *query()* – Based on the arguments passed to the method, the method creates a query and returns the dataset as an object *Cursor*,
- *insert()* – Based on the input arguments, it is possible to add data to the content provider database,
- *update()* – Method to update data,
- *delete()* – Method to erase data,
- *getType()* – Based on the specified URI address, the MIME type is returned ([30],
- *onCreate* - Method called after the creation of the class object and is used to initialize the class.

In addition to implementing the above methods, it is necessary to create *authority string* as the corresponding URI address.

It is also necessary to add the component information to the "AndroidManifest.xml" file with the *<provider>* tag. On the [31] page, you can find a complete example of building an application and component of type *content providers* together with a second application that uses the provided data. We can find more information on the [32] pages.

User interface

The user interface is the part of the application responsible for interacting with the user. A well-developed interface is essential for the user to operate the application correctly. We can distinguish between the part responsible for the graphic layout and the part responsible for handling events. In the Android system, layouts are created using XML files, which can be supported on the application code side. It is possible to create the layout only programmatically. The part responsible for handling events is implemented programmatically.

Layouts allow the arrangement of elements on the page, such as buttons, images, and text. They define the structure of the android user interface in the application, just like activities. The user interface in Android is built using two types of objects **ViewGroup** and **View**.

ViewGroup is an object type that can store other objects (ViewGroup and View). It is also the base class for containers and Layouts. The base parameters for layouts are also defined in this class (*ViewGroup.LayoutParams*) [25].

The View class is the base class for widgets, providing a set of basic properties and listeners to respond to events.

In the Android Studio environment, we have an editor that visually supports the creation of the graphical interface Figure no. 4.1. In particular, we can extract the following elements [24]:

1. *Palette* – Contains various views and view groups that you can drag into your layout.
2. *Component Tree* – Shows the hierarchy of components in your layout.
3. *Toolbar* – Click these buttons to configure your layout appears in the editor and change layout attributes.
4. *Design editor* – Edit your layout in Design view, Blueprint view, or both.
5. *Attributes* – Controls for the selected view's attributes.
6. *View mode* – View your layout in either: Code, Design, or Split.
7. *Zoom and pan controls* – Control the preview size and position within the editor.

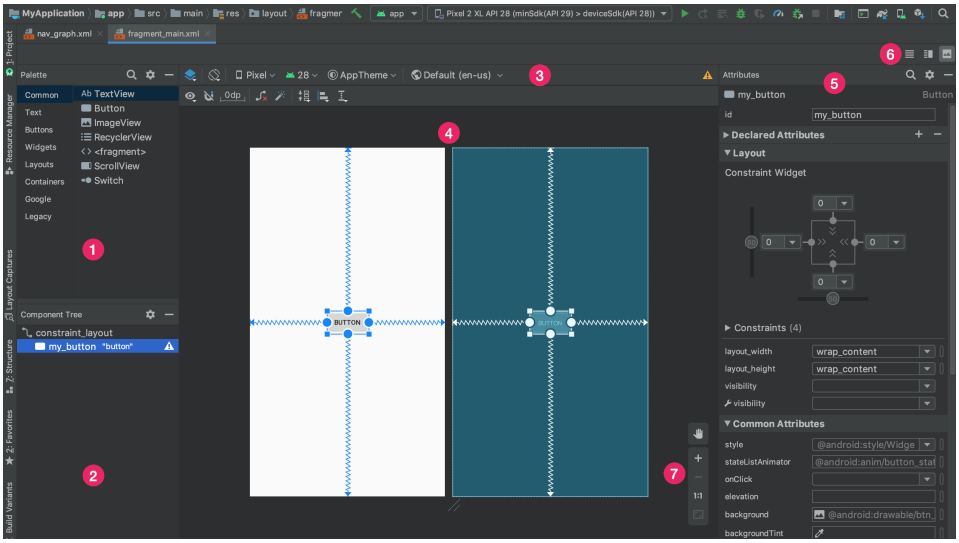


Figure 4.1: Layout Editor [24]

4.1 Create a layout

Layout files are typical resources and, according to the accepted file and directory structure in the Android project, are stored in the resources directory *res* in the subdirectory *res/layout*. In addition, the system allows for variant layout files, which will enable the layout to be defined according to different screen sizes. During operation, the system will select the layout appropriate for the screen's type or orientation. If none is available, the default one from the directory *res/layout* will be selected. Listing no. 4.1 shows the variant catalogues for portrait (default) and landscape screens for two types of devices,

```
1 res/layout/main_activity.xml                # Default
2 res/layout-land/main_activity.xml           # Default in landscape
3 res/layout-sw600dp/main_activity.xml        # For "7 tablets
4 res/layout-sw600dp-land/main_activity.xml   # For "7 tablets in landscape
```

Listing 4.1: Example of a variant directory structure with layouts

The Android Studio environment allows layout files to be added in several ways. The most common method is to select a particular layout directory and add a resource file via the right-click context menu "**New → Layout Resource File**".

The next step is to open the newly created file in *Layout Editor*. The selection of the appropriate Layout type depends on the application's requirements. The Android API provides the following types of layouts (subclass of ViewGroup):

- LinearLayout – arranges other views either horizontally in a single column or vertically in a single row, depending on parameter *android:orientation*
- RelativeLayout – displays child views in relative positions. The position of each view can be specified as relative to the parent element or in positions relative to the parent area.

- FrameLayout – is used to block off an area on the screen to display a single element. In general, a FrameLayout should be used to hold a single child view.
- GridLayout – places its children in a rectangular grid.
- RecyclerView – displays the list items; each list item has a layout, which can be one of the predefined ones, or you can create your own. The use of RecylerView forces the use of an adapter on the code side.
- ConstraintLayout – default layout for newly created applications, allows creating large and complex layouts with a flat hierarchy of views. Similar to RelativeLayout, but more flexible

The topic of layout creation is very extensive. In the documentation, you can find practical advice on properly creating a Layout [26] in the next subsection. We will focus exclusively on *ConstrantLayout* [27].

4.1.1 ConstraintLayout

This type of layout is the default layout. A correctly defined element requires two constraints to be defined. It is important to note that when using visual *Layout Editor*, the constraint values are often not defined at the beginning.

Various types of constraints that you can use [26]:

- Relative positioning – allows you to define a position in relation to another object,
- Margins – enforcing the margin as a space between the target and the source side,
- Centering positioning and Bias – centring object, We can also define Bias, which determines the offset ratio,
- Circular positioning – widget centre relative to another widget centre, at an angle and a distance,
- Visibility behaviour – allows you to handle widgets as *View.GONE*
- Dimension constraints – define minimum and maximum sizes for the ConstraintLayout itself
- Chains – provide group-like behaviour in a single axis (horizontally or vertically). The other axis can be constrained independently.

In addition to a visual editor, Android Studio provides typical layouts for many applications and code. An example of such a typical layout is the login form. To add it, right-click on the layout directory "→ **New** → **Activity (or Fragment)** → **Login Activity (or Login Fragment)**". The environment will automatically create the necessary entries and enter the required information into the AndroidManifes.xml file.

The generated layout can be found in Listing no. 4.2. The created Layout consists of one object of type *ViewGroup:ConstraintLayout*, and four view: two EditText, a Button, and a ProgressBar are of type View. Figure no. 4.2 shows the created view.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:app="http://schemas.android.com/apk/res-auto"
4      xmlns:tools="http://schemas.android.com/tools"
5      android:id="@+id/container"
6      android:layout_width="match_parent"
7      android:layout_height="match_parent"
8      android:paddingLeft="@dimen/activity_horizontal_margin"
9      android:paddingTop="@dimen/activity_vertical_margin"
10     android:paddingRight="@dimen/activity_horizontal_margin"
11     android:paddingBottom="@dimen/activity_vertical_margin"
12     tools:context=".ui.login.Login">
13
14     <EditText
15         android:id="@+id/username"
16         android:layout_width="0dp"
17         android:layout_height="wrap_content"
18         android:layout_marginTop="96dp"
19         android:autofillHints="@string/prompt_email"
20         android:hint="@string/prompt_email"
21         android:inputType="textEmailAddress"
22         android:selectAllOnFocus="true"
23         app:layout_constraintEnd_toEndOf="parent"
24         app:layout_constraintStart_toStartOf="parent"
25         app:layout_constraintTop_toTopOf="parent" />
26
27     <EditText
28         android:id="@+id/password"
29         android:layout_width="0dp"
30         android:layout_height="wrap_content"
31         android:layout_marginTop="8dp"
32         android:autofillHints="@string/prompt_password"
33         android:hint="@string/prompt_password"
34         android:imeActionLabel="@string/action_sign_in_short"
35         android:imeOptions="actionDone"
36         android:inputType="textPassword"
37         android:selectAllOnFocus="true"
38         app:layout_constraintEnd_toEndOf="parent"
39         app:layout_constraintStart_toStartOf="parent"
40         app:layout_constraintTop_toBottomOf="@+id/username" />
41
42     <Button
43         android:id="@+id/login"
44         android:layout_width="wrap_content"
45         android:layout_height="wrap_content"
46         android:layout_gravity="start"
47         android:layout_marginTop="16dp"
48         android:layout_marginBottom="64dp"
49         android:enabled="false"
50         android:text="@string/action_sign_in"
51         app:layout_constraintBottom_toBottomOf="parent"
52         app:layout_constraintEnd_toEndOf="parent"
53         app:layout_constraintStart_toStartOf="parent"
54         app:layout_constraintTop_toBottomOf="@+id/password"
55         app:layout_constraintVertical_bias="0.2" />
56
57     <ProgressBar
58         android:id="@+id/loading"
59         android:layout_width="wrap_content"
60         android:layout_height="wrap_content"
61         android:layout_gravity="center"
62         android:layout_marginTop="64dp"
63         android:layout_marginBottom="64dp"
64         android:visibility="gone"
65         app:layout_constraintBottom_toBottomOf="parent"
66         app:layout_constraintEnd_toEndOf="@+id/password"
67         app:layout_constraintStart_toStartOf="@+id/password"
68         app:layout_constraintTop_toTopOf="parent"
69         app:layout_constraintVertical_bias="0.3" />
70
71 </androidx.constraintlayout.widget.ConstraintLayout>
```

Listing 4.2: Example of Login layout

The Layout file follows the syntax of XML files. Hence there are characteristic elements, e.g. line no. 1 contains the obligatory definition of an XML file. Each object definition starts with the object class name preceded by "<" and ends with the class name as well, but is preceded by "</" and ends with ">".

It can be seen that the positions of the individual widgets are defined in relation (link) to other objects. Let us examine the definition of the button line by line 4.3. It is worth noting that some properties are common regardless of the type of object (lines 2–9), while lines 10–14 contain layout-specific properties *Constraint Layout*:

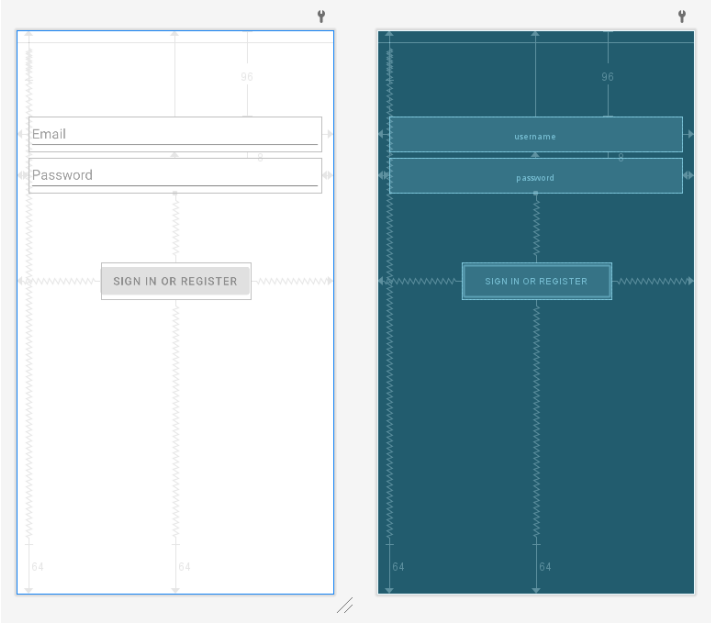


Figure 4.2: Example Login Layout

1. Object type definition – Button
2. Object identifier
3. Object width
4. Height of object
5. Gravity defines how the component should be placed in its cell group
6. Defines additional space from the top of the component
7. Defines additional space from the bottom of this component
8. The component is not active. Its state can be changed programmatically. In this case, the Login button will be activated when the user enters the login data.
9. Define the text to be displayed on the button. It is recommended that all constants are defined in the string.xml file of the application. In this case, the name is defined in the constant *action_sign_in*.
10. Align the bottom of an object to the bottom of another object.
11. Align the right (end) part of an object to the right part of another object.
12. Align the left(initial) part of an object to the left part of another object.
13. Align the top part of an object to the bottom part of another object.

14. Position an object along the horizontal axis, considering the bias value.

```
<Button
    android:id="@+id/login"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="start"
    android:layout_marginTop="16dp"
    android:layout_marginBottom="64dp"
    android:enabled="false"
    android:text="@string/action_sign_in"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/password"
    app:layout_constraintVertical_bias="0.2" />
```

Listing 4.3: Definition of button placement

Android provides a great deal of scope for shaping the appearance and positioning of objects, but learning all of them at an early stage is difficult. It should be added, however, that many attributes defining the location of individual objects and their behaviour coincide with the principles of creating interfaces, e.g. in web systems or creating applications in high-level languages. A separate issue is that different teams often handle the design, ensuring graphically consistent layouts and correct interaction with the user. The aforementioned templates or built-in graphical user interface tools are recommended for beginners.

4.2 Material Design

When designing a layout, it is worth (and even should) using the design pattern proposed by Google Material Design [34]. It was introduced in 2014 and is being developed all time. The 3rd version of this pattern [33] has now been introduced. **Material Design** was developed for Android and web apps. It can also be used for other systems. The introduction of **Material Design** aimed to standardise the user experience across all platforms and device sizes. This is done by defining guidelines for:

- principles of three-dimensionality (appropriate depths, shadows)
- the set of components – e.g. Cards, Dialogs, Menus, Lists, Sliders and many others
- colour sets
- font set
- icon set, or how to create your icons
- definitions of common shapes, e.g. rectangles and how to describe roundings
- animations
- interaction

The description page **Material Design** [34] provides, in addition to a detailed description of the listed elements, many tools to create interfaces quickly. For popular applications among interface designers (e.g. **Figma** [35]), documents and tools for the

rapid creation of interfaces are provided. It should be added that, despite standardisation, individual applications differ. The user mainly uses a common way of interaction by operating the applications with identical gestures or visually similar widgets. It should be emphasised that by using a design pattern to create the interface, users use the application in a very similar way and thus quickly and easily "learn" how to use it by replicating the already familiar interaction.

4.2.1 Interface development tools

The main page describing Material Design issues contains several tools for faster interface creation. If you want to build an interface correctly, it is necessary to familiarise yourself with the information contained there. Many documents are available describing the principles of interface construction, implementation details and, above all, good practices.

To bring the issue of interface construction a bit closer, we will present a component often used in applications. It is used to attractively give a group of objects or an ordered list. The component **Cards** is described in detail on page `https://material.io/components/cards/android#card`.

Cards can, in addition to presenting a title and short content, present multimedia elements and include an action, which most often serves as a transition to a more detailed presentation of an item. Figure no. 4.3 contains the presentation of the structure of the elements in the component.

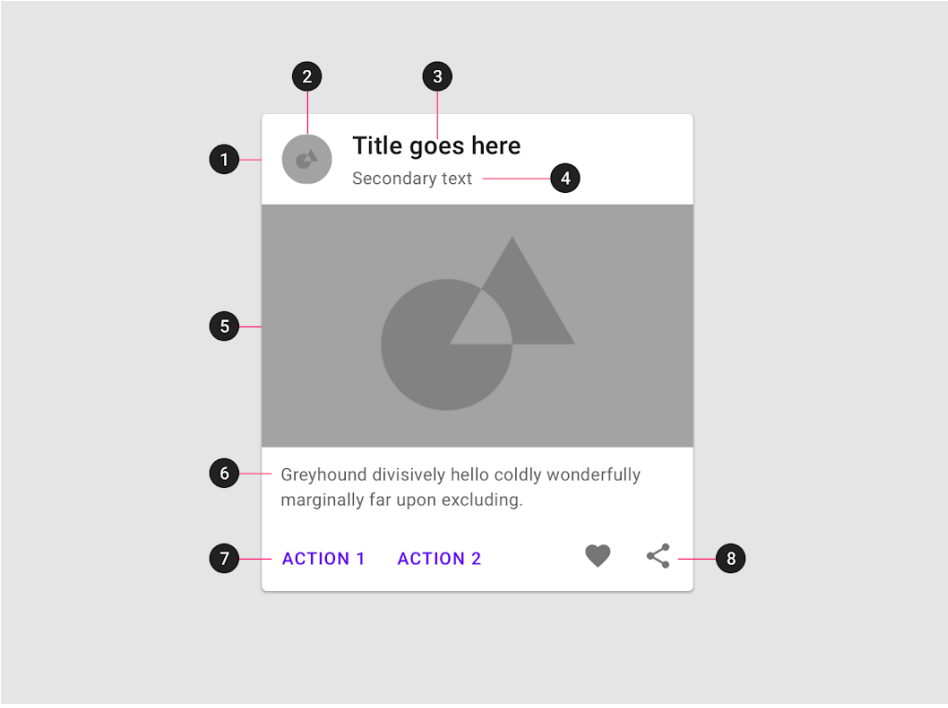


Figure 4.3: Structure of the Cards component [34]

1. Container – Card containers hold all card elements, and their size is determined by the space those elements occupy. The container expresses card elevation.
2. Thumbnail [optional] – Cards can include thumbnails to display an avatar, logo, or icon.
3. Header text [optional] – Header text can include things like the name of a photo album or article.
4. Subhead [optional] – Subhead text can include text elements such as an article byline or a tagged location.
5. Media [optional] – Cards can include a variety of media, including photos, and graphics, such as weather icons.
6. Supporting text [optional] – Supporting text includes text like an article summary or a restaurant description.
7. Buttons [optional] – Cards can include buttons for actions.
8. Icons [optional] – Cards can include icons for actions.

Listing no. 4.4 contains an example of layout code. The definition **Cards** starts at line no. 7 and refers to class *com.google.android.material.card.MaterialCardView*, so that we can use object-specific settings in the code. (The listing is on the next page.)

```xml
<?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"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.google.android.material.card.MaterialCardView
        android:id="@+id/card"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="8dp">

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical"
            android:background="@color/design_default_color_secondary">

            <!-- Media -->
            <ImageView
                android:layout_width="match_parent"
                android:layout_height="194dp"
                app:srcCompat="@drawable/Erasmus"
                android:scaleType="fitCenter"
                android:contentDescription="@string/content_description_media"
                />

            <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:orientation="vertical"
                android:padding="16dp">

                <!-- Title, secondary and supporting text -->
                <TextView
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:text="@string/title"
                    android:textAppearance="?attr/textAppearanceHeadline6"
                    />

                <TextView
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_marginTop="8dp"
                    android:text="@string/secondary_text"
                    android:textAppearance="?attr/textAppearanceBody2"
                    android:textColor="#004D40" />

                <TextView
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_marginTop="16dp"
                    android:text="@string/supporting_text"
                    android:textAppearance="?attr/textAppearanceBody2"
                    android:textColor="#004D40" />

            </LinearLayout>

            <!-- Buttons -->
            <LinearLayout
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_margin="8dp"
                android:orientation="horizontal">

                <com.google.android.material.button.MaterialButton
                    style="?attr/borderlessButtonStyle"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_marginEnd="8dp"
                    android:text="@string/action_1"
                    android:textColor="#5E35B1" />

                <com.google.android.material.button.MaterialButton
                    style="?attr/borderlessButtonStyle"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:text="@string/action_2"
                    android:textColor="#5E35B1" />
            </LinearLayout>

        </LinearLayout>

    </com.google.android.material.card.MaterialCardView>
</LinearLayout>
```

Listing 4.4: Example of a layout for a component (Cards)

Based on the layout code shown above, we obtain the view in Figure no. 4.4. In addition to the layout view, the depicted figure also contains a "blueprint". This representation of the view is very helpful, as it includes information on the boundaries of individual objects, obscured or hidden objects.

Figure 4.4: Example use of the component (Cards)

4.2.2 Color

Important for each application is the appropriate colour scheme. According to the philosophy **Material Design**, they should be vibrant but not bright. Each application can choose these at its discretion. However, it should stick to the guidelines. It is also essential that the choice of colours is appropriate for readability and provides adequate contrast between text and background. Another aspect is the selection of colours for the accessibility of the application.

Two main colours are used to create colour palettes for the app. The primary one is the base for creating further variations of it and is labelled 500. Based on this, colours labelled from 50–900 should be generated, where those from 50–400 are lighter than it, and those from 600–900 are darker.

The second (secondary) colour should stand out strongly from the primary colour and be used to highlight important interactive design elements, such as action buttons or links. It should also come from the secondary palette (marked with an A in the figure above) and be more saturated.

Light Blue 50	#E1F5FE
100	#B3E5FC
200	#81D4FA
300	#4FC3F7
400	#29B6F6
500	#03A9F4
600	#039BE5
700	#0288D1
800	#0277BD
900	#01579B
A100	#80D8FF
A200	#40C4FF
A400	#00B0FF
A700	#0091EA

Cyan 50	#E0F7FA
100	#B2EBF2
200	#80DEEA
300	#4DD0E1
400	#26C6DA
500	#00BCD4
600	#00ACC1
700	#0097A7
800	#00838F
900	#006064
A100	#84FFFF
A200	#18FFFF
A400	#00E5FF
A700	#00B8D4

Teal 50	#E0F2F1
100	#B2DFDB
200	#80CBC4
300	#4DB6AC
400	#26A69A
500	#009688
600	#00897B
700	#00796B
800	#00695C
900	#004D40
A100	#A7FFEB
A200	#64FFDA
A400	#1DE9B6
A700	#00BFA5

Figure 4.5: Examples of colour sets for apps proposed by Google in 2014 [34]

Google has provided a tool [36] that assists in selecting an appropriate colour scheme and includes detailed information on the availability of the chosen colour variant.

4.2.3 System icons

An important issue in the application is using appropriate symbols and icons to include marking actions or highlighting the functions of individual widgets. The tool provided at `https://fonts.google.com/icons` [37] allows one to search from more than 2,500 glyphs for a fitting symbol (icon) with its styling. Symbols are available in three styles and four adjustable variable font styles (fill, weight, genre and optical size)

Sensors

Android mobile devices are usually equipped with several different sensors to determine the device's movement, orientation or measurements of environmental conditions such as pressure or temperature. The quality and type of sensors built into the device depending on the model. Sensors enable new ways of interacting with the user (compared to desktop devices). They allow more convenient use of the device and provide new possibilities. Through the use of a range of sensors and the fusion of their data, applications can deliver specific content based on the user's current context. An example of such an application that has achieved considerable success is the game **Pokémon GO** [38].

Sensor support in Android they are divided into two types. Sensors are handled through a dedicated API, and sensors we can take with a common framework Sensors. This framework supports three main categories of sensors:

- **Motion sensors**, are used to measure the movement of a device. These sensors include accelerometers, gravity sensors, gyroscopes and rotation vector sensors.
- **Environmental sensors** measure various environmental conditions such as ambient air temperature and pressure, lighting and humidity. These sensors include barometers, photometers (light sensors) and thermometers.
- **Position sensors**, measure the physical position of a device. This category includes magnetometers (geomagnetic field sensors) and proximity sensors.

With the dedicated API, we support sensors such as :

- device camera,
- fingerprint sensor,
- microphone,
- GNSS modules (e.g. GPS module).

5.1 Sensor Framework

Sensor Framework is a set of classes and methods that defines sensor types, reading their properties such as maximum range or energy requirements. It also allows to specify of the minimum data acquisition frequency, register and de-register feedback methods for reading values and changes in sensor accuracy.

The framework consists of four main classes:

- *SensorManager*: Allows us to create a custom instance of the system service to access sensors. We can specify the type of embedded sensors, their properties and register and unregister methods to retrieve sensor values.
- *Sensor*: A class representing a single sensor provides methods to specify the sensor's capabilities.
- *SensorEvent*: Represents information about a sensor event. This event contains the raw sensor data, sensor type, data accuracy and a timestamp for the event.
- *SensorEventListener*: A programming interface, allowing event handlers to receive notifications of new sensor data or information about a change in sensor accuracy.

```
1  class SensorActivity : Activity(), SensorEventListener{
2      private lateinit var sensorManager: SensorManager
3      private var mSensor: Sensor? = null
4
5      public override fun onCreate(savedInstanceState: Bundle?){
6  ...
7          sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
8
9        //In this case we use the light sensor
10         mSensor = sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT)
11
12     }
13
14     override fun onAccuracyChanged(sensor: Sensor, accuracy: Int){
15         // Do something here if sensor accuracy changes.
16     }
17
18     override fun onSensorChanged(event: SensorEvent){
19         // The light sensor returns a single value.
20         val sensor_value = event.values[0]
21         // Do something with this sensor value.
22     }
23
24     override fun onResume(){
25         super.onResume()
26         ...
27         mSensor?.also{ sensor ->
28             sensorManager.registerListener(this, sensor, SensorManager.SENSOR_DELAY_NORMAL)
29         }
30     }
31
32     override fun onPause(){
33         super.onPause()
34         sensorManager.unregisterListener(this)
35     }
36 }
```

Listing 5.1: Skeleton using the Sensor Framework

The example code in Listing no. 5.1 contains complete code for reading one sensor. In line of code no. 7, we refer to the system service and access our instance of the object of class *SensorManager*, with its use in line no. 10, we read the values of the selected sensor. In this case, we read the light sensor, which is one-dimensional. Table no. 5.1 shows the possible sensors. In line no. 1 (class definition), we refer to the interface *SensorEventListener*, which acts as a callback and provides two methods:

1. *onAccuracyChanged(sensor: Sensor, accuracy: Int)* – this method is called when the accuracy of the sensor changes.
2. *onSensorChanged(event: SensorEvent)* – this method is called when the sensor value changes. The value is passed as an object in the form of a three-dimensional array of class *SensorEvent*. We read the actual data values from the specified table cell. If the sensor is one-dimensional, then we read from the table's first cell. In the case of sensors, e.g. an accelerometer, we read the following 3 values from the table, and these represent the individual axes (X, Y, Z)

The last step to read out the sensor data values is registering them. If we want to read the values continuously, it is necessary that the logging takes place according to the correct activity life cycle. For this reason, logging should be performed in method *onResume()*. The method *sensorManager.registerListener (this, sensor, SensorManager.SENSOR_DELAY_NORMAL)* contains three parameters: the first parameter specifies the callback methods (in this case, it uses the global for the listener class, hence the keyword *this*), the second parameter specifies the sensor object and the last parameter is the refresh rate. It is also necessary to unregister the sensor listener, which should be done in method *onPause()*.

Depending on our needs, we can use one of the 4 defined refresh rates [39]

- *SENSOR_DELAY_FASTEST* – get sensor data as fast as possible (0 ms)
- *SENSOR_DELAY_GAME* – rate suitable for games (20 ms)
- *SENSOR_DELAY_NORMAL* – rate (default) suitable for screen orientation changes (60 ms)
- *SENSOR_DELAY_UI* – rate suitable for the user interface (200 ms)

The refresh times indicate that data will be available no faster than the value provided. Much depends on the phone model itself and the processor load. Refresh rate *SENSOR_DELAY_FASTEST* means that data is downloaded as often as the system can. It should also be added that a higher refresh rate results in higher energy consumption by having to execute the method service *onSensorChanged()* more often. In addition, the method should only be used to retrieve and retain data. A separate mechanism should be set up for processing so that the possibility of retrieving data through pending subsequent events from the sensors is not blocked. Therefore, frequency reduction should always be considered to save energy and thus reduce performance hassles.

Tab. 5.1: *Sensor types supported by the Android platform [39]*

Sensor	Used for
TYPE_ACCELEROMETER	Motion detection (shake, tilt, and so on).
TYPE_AMBIENT_TEMPERATURE	Monitoring air temperature.
TYPE_GRAVITY	Motion detection (shake, tilt, and so on).
TYPE_GYROSCOPE	Rotation detection (spin, turn, and so on).
TYPE_LIGHT	Controlling screen brightness.
TYPE_LINEAR_ACCELERATION	Monitoring acceleration along a single axis.
TYPE_MAGNETIC_FIELD	Creating a compass.
TYPE_ORIENTATION	Determining device position.
TYPE_PRESSURE	Monitoring air pressure changes.
TYPE_PROXIMITY	Phone position during a call.
TYPE_RELATIVE_HUMIDITY	Monitoring ambient humidity (relative and absolute), and dew point.
TYPE_ROTATION_VECTOR	Motion and rotation detection.
TYPE_TEMPERATURE	Monitoring temperatures.

5.2 Location

Presenting the current position of a device is one of the main advantages of a mobile device. It provides the possibility of giving data based on the context of the work. The current determination of the position is used in many applications, ranging from navigation, searching for points of interest (POI), or checking, for example, weather conditions in a given area. The device's position (user) is sensitive data, so it is necessary to provide the application with permission to determine the position.

Currently, the most recommended approach to determining position is to use the Location API available in Google Play services. It uses a mechanism to determine position based on data from different providers, e.g. GNSS (GPS) module, WiFi module, and Bluetooth. The mechanism **Fused Location Provider** allows the use of information from previous queries, thus optimising energy consumption and speeding up position determination. The Android system allows you to determine whether the application requires an exact or approximate position. The use of location in an application requires several steps:

1. Definition of the location type
2. Requesting permission to locate the device
3. Fetching the location (last known, cyclic download of location)
4. Using the location, e.g. to show a point on the map

In the system, the location can be defined as coarse or as fine. From API level 29 (Android 10), it is still necessary to determine whether the location is of type **Foreground location** or **Background location**. Listing no. 5.2 shows how permissions are declared in the manifest file.

```
<manifest ... >
...
  <!-- Always include this permission -->
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

  <!-- Include only if your app benefits from precise location access. -->
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
  <!-- Recommended for Android 9 (API level 28) and lower. -->
  <!-- Required for Android 10 (API level 29) and higher. -->
...
  <!-- Required only when requesting background location access on
       Android 10 (API level 29) and higher. -->
  <uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
...
  <application...>
    <service
      android:name="MyNavigationService"
      android:foregroundServiceType="location" ... >
    </service>
  </application>
...
</manifest>
```

Listing 5.2: Defining permissions in the AndroidManifest.xml file

When requesting permissions, i.e. asking the user for permission, the rule is that we ask for permissions the first time they are used. In earlier versions, the user allowed permissions on installation, which, with many permissions, was incomprehensible to the user. He accepted them all with one permission. Some applications abuse permissions in this way. The mechanism for requesting permissions will be presented in the following

subsection. A detailed mechanism for requesting permissions for a location is shown at `https://developer.android.com/training/location/permissions` [40].

The third step is to retrieve the location. Here, we can consider the case when location information is needed once in the application without refreshing it during continuous use. In this case, we can use method *getLastLocation()* or *getCurrentLocation()* . We then use Google Play Services, so it is necessary to add a library usage declaration to **build.gradle** (Listing no. 5.3)

```
1 apply plugin: 'com.android.application'
2
3 ...
4
5 dependencies{
6     implementation 'com.google.android.gms:play-services-location:21.0.0'
7 }
```

Listing 5.3: Defining Google Play location services

The next step is to create your Provider instance (Listing no. 5.4) and to use a method in your application code that retrieves items from the system Provider (Listing no. 5.5).

```
1 private lateinit var fusedLocationClient: FusedLocationProviderClient
2
3 override fun onCreate(savedInstanceState: Bundle?){
4     // ...
5
6     fusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
7 }
```

Listing 5.4: Using fused location provider

```
1 fusedLocationClient.lastLocation
2         .addOnSuccessListener{ location : Location? ->
3             // Got last known location. In some rare situations this can be null.
4         }
```

Listing 5.5: Get location from provider

Retrieving the last known location is the fastest, but in some situations, the location may not be obtained, and an appropriate procedure should be implemented.

In the case where we want to receive cyclic location information, it is necessary to define return methods to receive notifications, and to start the retrieval of data accordingly according to the life cycle of the activity (or other component).

Listing no. 5.6 shows a fragment of an activity in which the location is cyclically updated [41]. The location information is cyclically received in the callback method *onLocationResult*, whose body is defined in lines 13–18. To receive this information, it is necessary, in addition to the previously mentioned permissions, to trigger the update of the location. In this example, this is contained in its own method *startLocationUpdate()* (code lines 28–32).It refers to one of the implementations *requestLocationUpdates*, where we specify the location conditions(*locationRequest*), the object to handle the location information event (*locationCallback*) and the thread in which the event is to be called. In this case, we use the main thread and thus can directly refer to objects on the GUI. This method is called the activity lifecycle method **onResume()**. The start of the listener in this code depends on a logical variable *requestingLocationUpdates*. This provides the possibility to control the application. The termination of location event retrieval is implemented in method *stopLocationUpdates()*, which can be called at any time and is always called when the activity is no longer visible on the screen.

```
private lateinit var locationCallback: LocationCallback
private lateinit var locationRequest: LocationRequest

...
override fun onCreate(savedInstanceState: Bundle?){
...
    locationRequest = LocationRequest.create()?.apply{
        interval = 10000
        fastestInterval = 5000
        priority = LocationRequest.PRIORITY_HIGH_ACCURACY
    }
    locationCallback = object : LocationCallback(){
        override fun onLocationResult(locationResult: LocationResult?){
            locationResult ?: return
            for (location in locationResult.locations){
                // Update UI with location data
                ...
            }
        }
    }
}

override fun onResume(){
    super.onResume()
    if (requestingLocationUpdates) startLocationUpdates()
}

private fun startLocationUpdates(){
    fusedLocationClient.requestLocationUpdates(locationRequest,
            locationCallback,
            Looper.getMainLooper())
}

override fun onPause(){
    super.onPause()
    stopLocationUpdates()
}

private fun stopLocationUpdates(){
    fusedLocationClient.removeLocationUpdates(locationCallback)
}
```

Listing 5.6: Using fused location provider

5.3 Request permissions

The Android system isolates individual applications in a sandbox, thus increasing security. If an application uses external or system resources, appropriate permissions must be granted. Permissions are divided into two groups, normal permissions, which do not require the user's permission, and so-called dangerous permissions, which need direct permission to use. Dangerous rights are often related to access to sensitive data, such as location data. Ordinary and unsafe permissions must first be entered in file *AndroidManifest.xml* (e.g. Listing no. 5.2). The next step is to ensure an appropriate workflow according to Google's recommendations`https://developer.android.com/training/permissions/requesting#workflow_for_requesting_permissions`. A diagram showing how to request permissions is shown in Figure no. 5.1

The diagram contains eight steps:

1. Declaration of permissions on file *AndroidManifest.xml*
2. The GUI design should allow the user to allow a specific action associated with a permission request.
3. Wait for user response, do not force assignment of permissions without user intervention
4. When the user takes action, first verify that permissions are already granted. You should verify the permissions every time the invoked method needs permission.

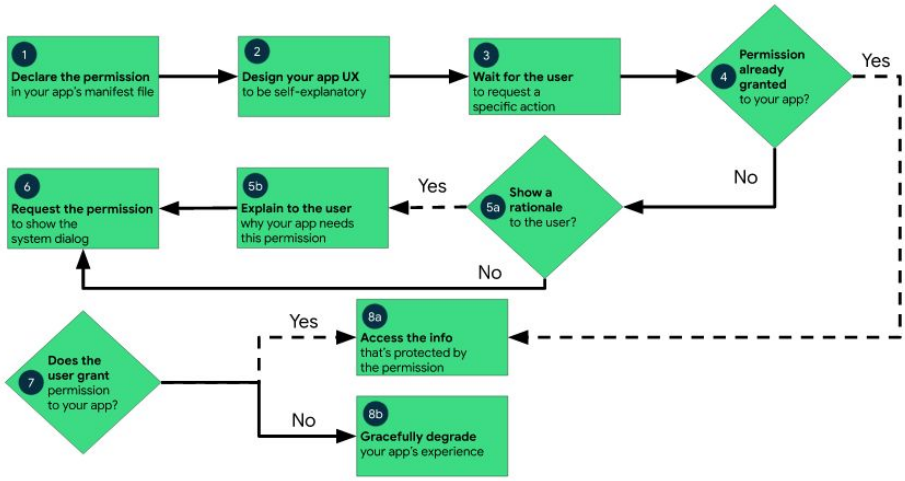


Figure 5.1: Workflow for declaring and requesting runtime permissions on Android [42]

5. Show the user information about why you need this permission. Usually, after the first refusal, we show this information. The first time you ask for this permission, you may not show this message.

6. Ask for the application request. The system will show the system permission information. When the user allows it, this ends the procedure.

7. Wait for the user's response.

8. In case of a first refusal, we show the user extended information why the permission is needed. In case of another refusal, we reduce the application's functionality, and if this is not possible and the permission is necessary, we close the application.

The described procedure is universal for all types of entitlements. Listing no. 5.7 shows an example code to create a request for permission and to handle cases of the first as well as the second refusal. The permission request is for access to a camera *Manifest.permission.CAMERA*. For readability, only the most relevant part is shown.

```
1  class MainActivity : AppCompatActivity(), ActivityCompat.OnRequestPermissionsResultCallback{
2
3      override fun onRequestPermissionsResult(
4              requestCode: Int,
5              permissions: Array<String>,
6              grantResults: IntArray
7      ){
8          if (requestCode == PERMISSION_REQUEST_CAMERA){
9              // Request for camera permission.
10             if (grantResults.size == 1 && grantResults[0] == PackageManager.PERMISSION_GRANTED){
11                 // Permission has been granted. Start camera preview Activity.
12                 layout.showSnackbar(R.string.camera_permission_granted, Snackbar.LENGTH_SHORT)
13                 startCamera()
14             } else{
15                 // Permission request was denied.
16                 layout.showSnackbar(R.string.camera_permission_denied, Snackbar.LENGTH_SHORT)
17             }
18         }
19     }
20
21     private fun showCameraPreview(){
22         // Check if the Camera permission has been granted
23         if (checkSelfPermissionCompat(Manifest.permission.CAMERA) ==
24                 PackageManager.PERMISSION_GRANTED){
25             // Permission is already available, start camera preview
26             layout.showSnackbar(R.string.camera_permission_available, Snackbar.LENGTH_SHORT)
27             startCamera()
28         } else{
29             // Permission is missing and must be requested.
30             requestCameraPermission()
31         }
32     }
33
34
35     private fun requestCameraPermission(){
36         // Permission has not been granted and must be requested.
37         if (shouldShowRequestPermissionRationaleCompat(Manifest.permission.CAMERA)){
38             // Provide an additional rationale to the user if the permission was not granted
39             // and the user would benefit from additional context for the use of the permission.
40             // Display a SnackBar with a button to request the missing permission.
41             layout.showSnackbar(R.string.camera_access_required,
42                     Snackbar.LENGTH_INDEFINITE, R.string.ok){
43                 requestPermissionsCompat(arrayOf(Manifest.permission.CAMERA),
44                         PERMISSION_REQUEST_CAMERA)
45             }
46
47         } else{
48             layout.showSnackbar(R.string.camera_permission_not_available, Snackbar.LENGTH_SHORT)
49
50             // Request the permission. The result will be received in onRequestPermissionResult().
51             requestPermissionsCompat(arrayOf(Manifest.permission.CAMERA), PERMISSION_REQUEST_CAMERA)
52         }
53     }
54
55     private fun startCamera(){
56 ...
57     }
58 }
```

Listing 5.7: Request permission example [43]

In the above code, the permission check is implemented in method *showCameraPreview()*, which is assigned to some button on the user interface.

Permission **Manifest.permission.CAMERA** is checked using the system method *checkSelfPermissionCompat(...)* (line 23) (step no. 4), if no permissions are assigned, the method *requestCameraPermission()* is called. First, it is checked whether it is necessary to display the extended information on the need to grant rights (line no. 37) (step no. 5a). Depending on the situation, the extended information is displayed (second attempt) or immediately (first attempt) the system window with the request for granting rights is displayed (line no. 43 or 51). This uses the method *requestPermissionsCompat(arrayOf(Manifest.permission.CAMERA), PERMISSION_REQUEST_CAMERA)*. After the user answers, the result is returned to the method *onRequestPermissionsResult(...)*. Assigning the rights ends the procedure and starts the method which handles the camera operation, while a message is displayed if the rights are not assigned.

When the camera action is attempted again, the checking procedure begins, with an extended message about the need to assign permissions displayed.

When multiple permissions need to be verified simultaneously in the application, we pass the matrix with the specified permissions and the so-called requestCode to the method *requestPermissionsCompat(..)* to distinguish the types of permissions. We verify them in method *onRequestPermissionsResult()* after the user responds and returns to the activity.

Data persistence

Most applications need to save data, e.g. data downloaded from the web or application settings. When wishing to save data, we should specify the data type to determine how it is saved. A separate issue is where the data is saved. Here we can save the data in internal memory, external memory (SD card) or an internet location (in particular, a cloud service). Table no. 6.1 summarises the possibilities for storing data in the device's memory. One possible way of accessing the data should be selected based on the data type.

Tab. 6.1: *Android storage capabilities [44]*

	Type of content	Access method	Permissions needed	Can other apps access?	Removed on app uninstall?
App-specific files	Files meant for your app's use only	From internal storage, getFilesDir() or getCacheDir()	Never	No	Yes
Media	Shareable media files (images, audio files, videos)	MediaStore API	Yes	Yes	No
Documents and other files	Other types of shareable content, including downloaded files	Storage Access Framework	None	Yes	No
App preferences	Key-value pairs	Jetpack Preferences library	None	No	Yes
Database	Structured data	Room persistence library	None	No	Yes

6.1 App preferences

Often the application adapts to the user by adjusting the colour scheme or setting default actions. It is also common to allow simple data to be stored that is entered into the application (e.g. login). Most often, this data is of a simple type. They are limited to single words or values. Therefore, a dedicated mechanism called **Shared Preferences** has been created for storing this type of data. It consists of storing data represented as a key-value. With the help of this mechanism, we can save simple types (String, Int, Long, Float, Boolean). By default, the data is stored in the device's internal memory in the application data directory (“**/data/data/PACKAGE_NAME/shared_prefs/PREFS_NAME.xml**”).

```
...
val sharedPref = activity?.getSharedPreferences(getString(R.string.preference_file_key), Context.MODE_PRIVATE)
...
//Read from Shared Preferences
val highScore = sharedPref.getInt(getString(R.string.saved_high_score_key),defaultValue)
...
//Write to Shared Preferences
with (sharedPref.edit()){
    putInt(getString(R.string.saved_high_score_key), newHighScore)
    apply() // commit() -- synchronously
}
```

Listing 6.1: Shared Preferences example

An example of the Shared Preferences API is shown in Listing no:6.1. First, we need to create an object, referring to the file **Shared Preferences** (Listing no. 2). With the object created, we can use a method that reads from this file, e.g. *getInt(...)*. We give it the name of the key we are looking for and the default value as the second parameter. If a key is in the file, the value assigned to it will be returned. If there is no key in the file, a default value will be given to prevent a potential error of missing value. Writing the value is similar, we use the method *putInt(...)*, the first argument is the name of the key, followed by the value. It will be added if the key does not exist in the file. If it does exist, it will be replaced by a new value. The last step is to confirm the save. We can do this synchronously (*commit()* or asynchronously *apply()*).

6.2 Room – Database

In the Android environment, **SQLite** was chosen as the base database for storing structured data. SQLite is a database without the need to run a separate RDBMS process. The contents of the database are stored in a single file. Database security is based on file system security. Each database in Android is stored in the private memory of each application in a dedicated directory (**"/data/data/PACKAGE_NAME/databases/DB_NAME"**). Initially, developers directly implemented all elements related to the database using the **android.database.sqlite** package. Several classes, including schema, contract class, SQL-helper class and queries, had to be defined and managed independently. With the proposal to create an application in line with "Android Architecture Components" [15] and the introduction of the framework ROOM, the use of the database in the application was greatly simplified. Figure no 6.1 shows the basic model "**Android Architecture Components**". The whole is based on the MVVM design pattern (to be presented in the next chapter), and one of its elements is repositories, including a database based on **SQLite**. Room framework allows for faster and error-free query creation. Support for integration with other elements is also important **Architecture components like LiveData**.

Room has three main components of Room DB:

- Entity
- Dao
- Database

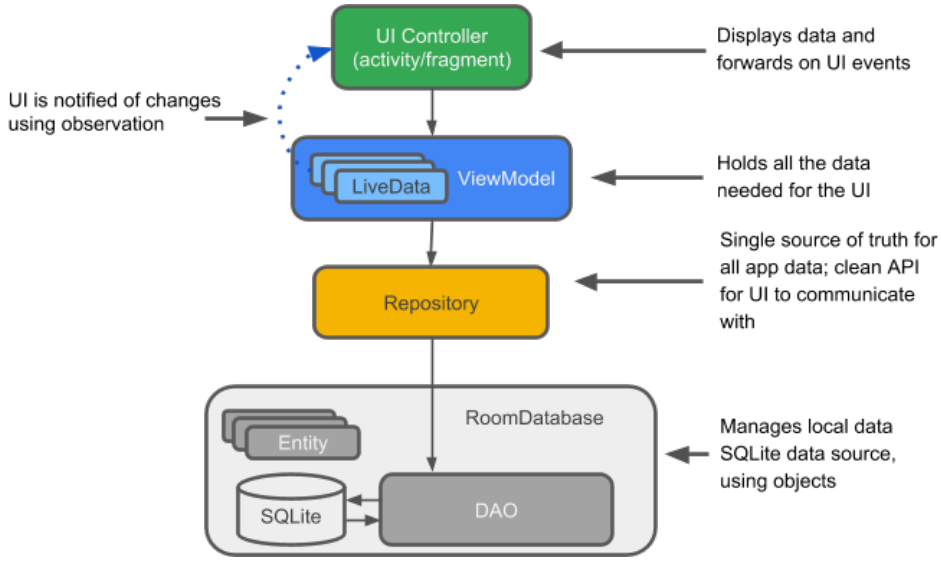


Figure 6.1: Diagram of Architecture Components [45]

6.2.1 Entity

Represents a table within the database. Room creates a table for each class that has *@Entity* annotation. The class fields correspond to columns in the table. Therefore, the entity classes tend to be small model classes that don't contain any logic. The use of annotations significantly speeds up code development. We have a list of possibilities on the webpage `https://developer.android.com/reference/androidx/room/package-summary#annotations`. The following are often used:

- *@PrimaryKeys* – as its name indicates, this annotation points to the primary key of the entity. *autoGenerate* — if set to true, then SQLite will generate a unique id for the column,
- *@ColumnInfo* – allows specifying custom information about column,
- *@Ignore* — field will not be persisted by Room, e.g. picture.

```
1  @Entity
2  data class Person (
3      @PrimaryKey (autoGenerate=true)
4      var uid : Int? = null
5
6      @ColumnInfo(name = "first_name")
7      var firstName: String
8
9      @ColumnInfo(name = "last_name")
10     var lastName: String
11 )
```

Listing 6.2: Entity exampl [45]

Listing no. 6.2 contains an example definition of a table. Line no. 1 defines that the class defined below is a table. At the same time, this is the definition of a class of type

data (Line no. 2). It contains 3 columns. Column *uid* is the primary key, and the system automatically generates its value (always ascending). The second column in the database will be named *first_name*, and when referring to the class, we will use the name *firstName*, similarly to the third column relating to the surname.

6.2.2 DAO

DAOs – Data Access Object – are responsible for defining the methods that access the database. In the initial SQLite, we use the Cursor objects. With Room, we don't need all the Cursor related code and can define our queries using annotations in the Dao class. **DAO** provides the methods that the rest of the app uses to interact with data in the tables.

```
1  @Dao
2  interface PersonDao{
3
4      // The conflict strategy defines what happens,
5      // if there is an existing entry.
6      // The default action is ABORT.
7      @Insert(onConflict = OnConflictStrategy.REPLACE)
8      fun insert(Person person)
9
10     // Update multiple entries with one call.
11     @Update
12     fun updatePersons(persons: List<Person>)
13
14     // Simple query that does not take parameters and returns nothing.
15     @Query("DELETE FROM person")
16     fun deleteAll()
17
18     // Simple query without parameters that return values.
19     @Query("SELECT * from person ORDER BY lastName ASC")
20     fun getAllPerons(): List<Person>
21
22     // Query with parameter that returns a specific person or persons.
23     @Query("SELECT * FROM person WHERE last_name LIKE :lastName ")
24     fun findPersons(lastName: String?):  List<Person>
25 }
```

Listing 6.3: @Dao exampl [45]

Listing no. 6.3 contains an example definition *@Dao*. Line no. 1 defines that the class defined below is of type DAO. It should be added that the class defining the queries is an interface. The class contains the definition of 5 methods representing queries to the database. The implementation of this interface is created automatically based on the SQL queries defined as *@Query*, the add (annotation *@Insert*) and update operations for the data*@Update* are also automatically created.

6.2.3 Database

It contains the database holder and is the main access point for the underlying connection to your app's persisted relational data. The implementation is an abstract class that extends class *RoomDatabase*, is decorated with annotation *@Database* and lists all defined tables. Objects are also defined in the body of the class **DAO**. The conditions that an annotated class must satisfy *@Database*:

- It must be an abstract class that extends RoomDatabase,
- Must specify a list of all entities within the annotation,
- It must contain an abstract method that references the annotated class *@DAO*,
- At runtime, we shall acquire an instance of **Database** by calling *Room.databaseBuilder()* or *Room.inMemoryDatabaseBuilder()*.

```
1  @Database(entities = arrayOf(Person::class), version = 1, exportSchema = false)
2  public abstract class PersonDatabase : RoomDatabase{
3
4      abstract fun personDao() : PersonDao;
5
6      private static PersonDatabase INSTANCE;
7      companion object{
8
9          @Volatile
10         private var INSTANCE: PersonDatabase? = null
11
12         fun getDatasetClient(context: Context) : PersonDatabase{
13             if (INSTANCE != null) return INSTANCE!!
14
15             synchronized(this){
16                 INSTANCE = Room
17                     .databaseBuilder(context, PersonDatabase::class.java, "person_database")
18                     .fallbackToDestructiveMigration()
19                     .build()
20
21                 return INSTANCE!!
22             }
23         }
24
25     }
26 }
```

Listing 6.4: @Database exampl [45]

Listing no. 6.4 contains an example definition **@Database**. According to the previously mentioned conditions, this code includes all required conditions. The example class is abstract (Line no. 2), Line no. 1 contains the entity information, Line no. 4 has the class information from *@Dao*. Access to the database instance is implemented via method *Room.databaseBuilder()* (Line no. 11). It should be added that in this class, we define the name of the database, it is "**person_database**", which is one of the arguments of the method *Room.databaseBuilder()*.

6.2.4 Repository – How manage Database

Having defined the elements that constitute access to the database from the framework **Room**, the last step is to use them. For this purpose, it is advisable to create a class **Repository** that stores access to the data, and through it, we can manage the database. Direct access is also possible, but such an architecture is not recommended.

```
class PersonRepository{
    companion object{

      var personalDatabase: PersonalDatabase? = null
        var personDao: PersonDao? = null

        fun initializeDB(context: Context) : PersonDatabase{
            return PersonDatabase.getDatasetClient(context)

        }

        fun insertData(context: Context, person: Person){

            personalDatabase = initializeDB(context)

            CoroutineScope(IO).launch{

                personDatabase!!.personDao().insert(person)
            }

        }

        fun getAllPersons(context: Context) : LiveData<List<Person>>?{

            personDatabase = initializeDB(context)
            return personDatabase!!.personDao().getAllPersons()

        }

    }


   ...
}
```

Listing 6.5: Repository exampl [45]

Creating a class **Repository** allows us to manage the database centrally. We have a collection of all available queries and helper classes.

The next step is to call the methods from the created class **Repository**. The framework **Room** is part of the model "**Android Architecture Components**", and using it with the MVVM pattern is recommended. Listing 6.6 shows the use of the previously created class *PersonRepository* in a class that is one of the elements of MVVM.

```

class PersonViewModel : ViewModel(){

    fun insertData(context: Context, person: Person){
       PersonRepository.insert(context, person)
    }

    fun getAllPersons(context: Context) : LiveData<List<Person>>?{
        return PersonRepository.getAllPersons(context)
    }

}
```

Listing 6.6: Using the repository class witch MVVM [45]

Design pattern MVVM

When creating applications, we rarely use single components. We often develop applications with multiple components sharing the same data between them. In earlier chapters, the life cycles of individual components were presented. From a data management point of view, the fundamental problem is to inform components about changing data. The previous one showed how to access data in particular data stored in a database. The initial pattern used is to implement actions that depend on the component. The component itself takes care of data updates. However, this pattern can result in poor code organisation and the appearance of bugs and is practically only effective for applications with single functionalities. When data changes in different activities, this approach is ineffective and can lead to delays and problems with displaying current data. Another approach is to use a pattern in which individual components react to changing data and are sensitive to the life cycle of other components. For Android, such a pattern is Model-View-ViewModel, abbreviated as MVVM, which is part of "**Android Architecture Components**" [15].

The model MVVM introduces three layers:

- Model – This layer is responsible for the abstraction of data sources. Model and ViewModel work together to retrieve and store data. Such a function can be performed by a database together with a framework **Room**.
- View – The function of this layer is to inform ViewModel about user actions. This layer observes ViewModel and does not contain any application logic.
- ViewModel- Deals with providing model data for the view layer and taking action on a triggered event from the view.

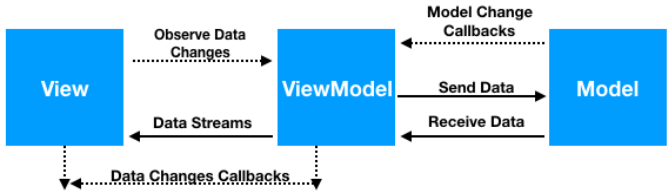


Figure 7.1: Android MVVM patern [16]

Its logic is minimised by using a data-binding strategy in the view layer. The code becomes more structured and open to modification, and testing is easier. The idea behind

the MVVM pattern is that the view layer observes (Observer pattern) the changing data in the model layer and reacts to the changes through a data binding mechanism.

The Android Jetpack introduces facilities for creating an application that follows the MVVM pattern, in particular, it provides classes:

- ViewModel,
- LiveData, MutableLiveData, MediatorLiveData,
- Lifecycle.

The below contains good practices for working with components in applications with the MVVM pattern.

UI controllers (activities and fragments) should be as simple as possible. They should not attempt to acquire their data. The UI controller is responsible for updating views when data changes or notifying ViewModel classes of user actions.

In classes that are **ViewModel**, it is recommended to observe objects using **LiveData**. This will allow the user interface to be informed of changes.

It would help if you wrote data-driven user interfaces where the controller **UI** is responsible for updating views when data changes. The ViewModel should be notified of user actions.

The data logic should be placed in class **ViewModel**. The ViewModel should serve as a link between the UI controller and the rest of the application. However, e.g. IO operations (e.g. download file) should not be implemented in the VM class. Instead, creating separate components that retrieve the data and return the result to the UI controller is recommended. In addition, they must be executed in a non-UI thread. It is recommended to use **data binding** so that the interface between views and the UI controller is transparent and the views become more declarative with a minimum of update code. To manage long-running tasks, use **Kotlin coroutines** [46], which can be executed asynchronously.

7.1 ViewModel

The main task of the class being **ViewModel** is to maintain information (caching) about the state values of objects on the user interface. It allows the state to persist regardless of the current activity and, when the user returns to the interface associated with a given **ViewModel**, to provide the current values. The second function **ViewModel** is to provide access to the application's business logic. In addition, **ViewModel** is responsible for handling events and forwarding them to other layers.

An example of an implementation was presented in the previous chapter in Listing no. 6.6. The example class extends class *ViewModel*. According to the adopted methodology, this class contains all data operations. Using *LiveData*, it receives all changes to the associated objects. Listing no. 7.1 shows the code belonging to the user interface layer, which uses the stored data through **ViewModel**. It is also important that many different UIs can access the same VM.

```
class MainActivity : AppCompatActivity(){

    lateinit var personViewModel: PersonViewModel
    lateinit var context: Context
...
    override fun onCreate(savedInstanceState: Bundle?){
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        context = this@MainActivity

        personViewModel = ViewModelProvider(this).get(PersonViewModel::class.java)

        btnAddPerson.setOnClickListener{
...
                personViewModel.insertData(context, person)
...
            }
        }

        btnGetPersons.setOnClickListener{
...

            loginViewModel.getAllPersons().observe(this, new Observer<User>(){
            @Override
            public void onChanged(@Nullable Person data){
                // update ui.

            }
        });
        }
    }
}
```

Listing 7.1: ViewModel Example

7.2 LiveData

Class **LiveData** – an observable data holder class allows you to create applications that are sensitive to the life cycles of components such as fragments or activities. It notifies the observable of changing data while observing lifecycle changes, updating only active observables. There are subclasses in LiveData that are useful for their properties when updating the UI [47]:

- LiveData – is immutable by default. Using LiveData, we can only observe the data and cannot set the data.
- MutableLiveData – subclass of LiveData. In MutableLiveData, we can observe and set the values using *postValue()* and *setValue()* methods (the former being thread-safe) so that we can dispatch values to any live or active observers.
- MediatorLiveData – can observe other LiveData objects, such as sources and react to their *onChange()* events. MediatorLiveData will give us control over when we want to perform an action in particular or to propagate an event.

Advantages of using **LiveData**:

- Removes the leaks caused by the interfaces/callbacks that send results to the UI thread.
- It de-couples tight integration between data, mediator, and the UI layers.
- Always up to date with the latest data
- Sharing resources, e.g. Extending LiveData

7.2.1 Using LiveData

We can illustrate the use of **LiveData** in the following steps, with reference in brackets to previously shown examples:

1. Create a LiveData instance in your ViewModel class to hold the data. (Listing no. 6.6, Line no. 8)
2. Set the data in LiveData. (Listing no. 6.5, Line no. 23–28)
3. Return the LiveData. (Listing no. 6.6, Line no. 8–10)
4. Observe the data with the help of the Observer() function.(Listing no. 7.1, Line no. 24–30)

Networking

When developing applications, there is often a need to use network connections. Network requests are used to download or update data. The connectivity of mobile devices is very large, and we can distinguish between media connections:

- Wi-Fi networks,
- cellular networks,
- Bluetooth,
- NFC.

For internet connections, the first two are most commonly used. When using networks, special attention should be paid to the aspect of energy saving as well as the use of an unmetered network (e.g. WiFi) to download significant portions of data.

The use of the network can generate considerable costs, as well as being a particular threat to the user's privacy, which is why permission to use the network is necessary, and the perform network operations and read network status in your application, your manifest must include the permissions Listing no. 8.1.

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

Listing 8.1: Define permissions

Before connecting, it is good practice to check that the device is connected to the Internet. Network checks must be performed before any download operation.

```
1 private fun isNetworkConnected(): Boolean{
2   val connectivityManager = getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
3   val activeNetwork = connectivityManager.activeNetwork
4   val networkCapabilities = connectivityManager.getNetworkCapabilities(activeNetwork)
5   return networkCapabilities != null &&  networkCapabilities.hasCapability(NetworkCapabilities.NET_CAPABILITY_INTERNET)
6 }
```

Listing 8.2: Checking network exampl [49]

Listing no. 8.2 contains a method that returns whether a network is available. Line no. 2 fetches an instance of **ConnectivityManager**, and with its help, we check the network properties, including whether the available networks are connected to the Internet (Line no. 5). This simple method will allow us to avoid trouble considerably and at the same time using it will allow us to inform the user about the need to connect to the network.

Performing network operations requires them to be completed in a separate thread not to burden the main thread. It is dedicated to the user interface. Currently, most applications use a given service's REST API [52] to retrieve data. Sometimes, however, it

is necessary to retrieve a file from the server in the traditional way. The HTTP/HTTPS protocol is then preferred. This makes it quite challenging to create communication with full support. This is why we most often use dedicated libraries to handle network connections. In the Android environment, we have many libraries that facilitate the creation of network connections, such as Retrofit, Volley, OkHttp (HTTP + HTTP /2). It is also possible to use a native solution, i.e. to use a class **HttpUrlConnection**. In the following subsections, a description of the use of the library **Retrofit** [48] and class **HttpUrlConnection** will be presented.

8.1 HTTP connections using HttpUrlConnection

Using **HttpUrlConnection**, we do not have to specify any dependencies, we have control over the connection process, but at the same time, we have to handle the connection properly, especially the cancellation of the connection. This often causes implementation problems. The downloaded data is passed as RAW to *InputStream*, and later decoding is necessary. We also need to ensure that the application runs in a separate thread. Listing no. 8.3 contains the activity to retrieve the web page data. In this case, it is a graphic image. In the example application, ensuring that the data is fetched in a separate thread is implemented through **Kotlin coroutines**. This mechanism allows for multitasking. To specify where the coroutines should run, Kotlin provides three dispatchers that you can use:

- Dispatchers.Main – Use this dispatcher to run a coroutine on the main Android thread. This should be used only to interact with the UI and perform quick work.
- Dispatchers.IO – This dispatcher is optimised to perform disk or network I/O outside the main thread.
- Dispatchers.Default – This dispatcher is optimised to perform CPU-intensive work outside the main thread.

In the example Listing no. 8.3, we wrapped our network call in the IO dispatchers (Line no. 19–21). Additionally, for the function to be executed in a separate thread, it must be marked as *suspend* [51] Line no. 32. Line no. 60 contains a reference to changing an object on the screen. Such changes must be executed in the main thread (i.e. the UI thread), the call to this method is preceded by /textitCoroutineScope(Dispatchers.Main).launch(). Using the **Coroutine** mechanism allows us to change the thread as required.

```
1
2  class MainActivity : AppCompatActivity(){
3      lateinit var button: Button
4      lateinit var imageView: ImageView
5
6
7     private val ioScope = CoroutineScope(Dispatchers.IO)
8      override fun onCreate(savedInstanceState: Bundle?){
9          super.onCreate(savedInstanceState)
10         setContentView(R.layout.activity_main)
11
12         button = findViewById(R.id.button)
13         imageView = findViewById(R.id.imageView)
14
15         val service = GithubApiService.Factory.create()
16
17         button.setOnClickListener{
18             if (isNetworkConnected()){
19                 ioScope.launch{
20                     downloadFile("http://ioscs.zut.edu.pl/fileadmin/image/Opensource.svg.png")}
21             }
22         }
23     }
24
25     private fun isNetworkConnected(): Boolean{
26         val connectivityManager = getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
27         val activeNetwork = connectivityManager.activeNetwork
28         val networkCapabilities = connectivityManager.getNetworkCapabilities(activeNetwork)
29         return networkCapabilities != null &&  networkCapabilities.hasCapability(NetworkCapabilities.
               NET_CAPABILITY_INTERNET)
30     }
31
32     suspend fun downloadFile(inputUrl:String){
33         val url = URL(inputUrl)
34         val httpClient = url.openConnection() as HttpURLConnection
35         httpClient.doInput=true
36         httpClient.connectTimeout = 5000
37         httpClient.readTimeout = 5000
38
39         if (httpClient.responseCode == HttpURLConnection.HTTP_OK){
40             try{
41                 val stream = BufferedInputStream(httpClient.inputStream)
42                 readStream(stream)
43
44             } catch (e: Exception){
45                 e.printStackTrace()
46             } finally{
47                 httpClient.disconnect()
48             }
49         } else{
50             Log.v("Error in Comuniaction","ERROR"+ httpClient.responseCode)
51         }
52     }
53
54     private fun readStream(inputStream: InputStream){
55         //data-dependent implementation of the reader
56
57         val bitmapImage = BitmapFactory.decodeStream(inputStream)
58
59         CoroutineScope(Dispatchers.Main).launch(){
60             imageView.setImageBitmap(bitmapImage)}
61     }
62 }
```

Listing 8.3: Example of use of the class HttpURLConnection

The class methods *HttpUrlConnection* allow us to define connection parameters, including the setting of times, the definition of authorisation or the use of different types of queries. However, this class, despite its range of possibilities, is less frequently used than other libraries, notably Retrofit.

8.2 HTTP connections using Retrofit

Retrofit is an Android and Java library that excels at retrieving and uploading structured data, such as JSON and XML. This library makes HTTP requests using OkHttp, another library from Square.

Downloaded data from the internet is very often serialised, e.g. in **XML**, **JSON** format. Retrofit can serialise and deserialise data using various libraries:

- Gson,
- Jackson,
- Moshi,
- Protobuf,
- Wire,
- Simple XML.

This gives the programmer many possibilities when choosing serialisation using this library. In addition, the library deals with thread handling.

Like the other libraries, it is first necessary to add information about it to the dependencies in file *build.gradle* in the app module (Listing no. 8.4), and second, we add the GSON library for format deserialisation **JSON**.

```
implementation 'com.squareup.retrofit2:retrofit:$VERSION$'
implementation 'com.squareup.retrofit2:converter-gson:$VERSION$'
```

Listing 8.4: Retrofit depedencies

Using **Retrofit**, we first declare the interface, which defines the HTTP method, address, arguments, and response type. At Line no. 11–19, an instance of the service is created that queries the defined URL. In this case, we use the API from page `https://api.github.com/`. The complete documentation can be found at `https://docs.github.com/en/rest`.

```
interface GithubApiService{

    @GET("search/users")
    fun search(@Query("q") query: String,
               @Query("page") page: Int = 1,
               @Query("per_page") perPage: Int = 20): List<Result>

    /**
     * Companion object to create the GithubApiService
     */
    companion object Factory{
        fun create(): GithubApiService{
            val retrofit = Retrofit.Builder()
                    .addConverterFactory(GsonConverterFactory.create())
                    .baseUrl("https://api.github.com/")
                    .build()

            return retrofit.create(GithubApiService::class.java);
        }
    }
}
```

Listing 8.5: Definig query using Retrofit

Listing no. 8.5 presents two example GET requests.

```
class SearchRepository(val apiService: GithubApiService){

    fun searchUsers(location: String, language: String): Lsit<Result>{
        return apiService.search(query = "location:$location language:$language")
    }

    fun searchUsers(username: String): List<Result>{
        return apiService.search(query = username)
    }
}
```

Listing 8.6: Repository example

The data that will be returned during a query is specific data models, which should conform to the provider-provided data model.

```
data class User(
        val login: String,
        val id: Long,
        val url: String,
        val html_url: String,
        val followers_url: String,
        val following_url: String,
        val starred_url: String,
        val gists_url: String,
        val type: String,
        val score: Double
)
data class Result (
       val total_count: Int,
       val incomplete_results: Boolean,
       val items: List<User>
)
```

Listing 8.7: Data classes – Retrofit example

The final step is to use the created classes. Listing no. 8.8 shows an example call to the search method. In line no. 1, we define the local object with the help we will make the query. Line no. 4 shows the method call *search* with the parameter. An asynchronous query is started by calling the method *enqueue*. The result will be returned in the method callbacks. In method *onReponse*, line no. 7–10, the received value will be returned. In case of a communication error, wrong request, or no response, it will be handled in method *onFailure*.

```
val service = GithubApiService.Factory.create()

button.setOnClickListener{
    val searchRequest = service.search("rmaciaszczyk")

    searchRequest.enqueue(object : Callback<List<Result>>{
         override fun onResponse(call: Call<List<Result>>, response: Response<List<Result>>){
             val userProperties = response.body()
         Log.v("Result{userProperties}")
         }


         override fun onFailure(call: Call<List<Result>>, t: Throwable){
             Log.i(MainActivity::class.simpleName, "on FAILURE!!!!")
         })
}
```

Listing 8.8: Call retrofit method

Retrofit allows the developer to focus on the purpose of the communication rather than its handling. MVVM supports this library, which makes it recommended for handling network connections, particularly retrieving data using queries **REST**.

Summary

This publication contains information about mobile programming applications in the most popular mobile operating system – Android. The publication has been prepared as support material for classes in the subject of **Mobile Application Development**. The publication can be used by lecturers, students and pupils during classes and by people who want to expand their knowledge of Android's programming basics. The book uses the official Android documentation along with some of the code. The timeliness of the included descriptions is as of June 2022. The development of the system means that some of the material may become outdated over time. In particular, a new approach to solving given issues may be proposed. Developing mobile applications for Android is possible with free software released under an open source licence – Android Studio. Knowledge of object-oriented programming and the basics of the Kotlin language is also recommended. These minor requirements mean that mobile app programming can be for everyone. It is also worth noting that the system's popularity is already so high that we can successfully write an **Android is everywhere**.

Bibliography

[1] Mobile Operating System Market Share Worldwide, https://gs.statcounter.com/os-market-share/mobile/worldwide, 03.2022

[2] Mobile Vendor Market Share Worldwide, https://gs.statcounter.com/vendor-market-share/mobile/worldwide, 03.2022

[3] Simon Kemp, DIGITAL 2022: GLOBAL OVERVIEW REPORT, https://datareportal.com/reports/digital-2022-global-overview-report, 26.01.2022

[4] OHA1, `http://www.openhandsetalliance.com/press_110507.html`

[5] OHA1, `http://www.openhandsetalliance.com/press_111207.html`

[6] OHA1, `https://pl.wikipedia.org/wiki/G1\_(telefon)`

[7] OHA1, `https://en.wikipedia.org/wiki/Android_version_history`

[8] Kotlin docs, `https://kotlinlang.org/docs/home.html`

[9] A modern programming language that makes developers happier, `https://kotlinlang.org/#why-kotlin`

[10] Mark L. Murphy, Elements of Kotlin, `https://commonsware.com/Kotlin/`, CommonsWare 2021

[11] Android source `https://source.android.com/`

[12] Android source `https://cs.android.com/`

[13] Android studio `https://developer.android.com/studio`

[14] Meet Android Studio, `https://developer.android.com/studio/intro`

[15] Guide to app architecture, `https://developer.android.com/topic/architecture`

[16] Activity, `https://developer.android.com/reference/android/app/Activity`

[17] App Manifest Overview, `https://developer.android.com/guide/topics/manifest/manifest-intro`

[18] Fragment lifecycle, `https://developer.android.com/guide/fragments/lifecycle`

[19] Navigation Editor, https://developer.android.com/guide/navigation/navigation-getting-started#nav-editor

[20] Pass data between destinations, https://developer.android.com/guide/navigation/navigation-pass-data

[21] Services overview, https://developer.android.com/guide/components/services

[22] Implementing the lifecycle callbacks, https://developer.android.com/guide/components/services#LifecycleCallbacks

[23] Bound services overview, https://developer.android.com/guide/components/bound-services

[24] Build a UI with Layout Editor, https://developer.android.com/studio/write/layout-editor

[25] ViewGroup, https://developer.android.com/reference/android/view/ViewGroup

[26] ConstraintLayout, https://developer.android.com/reference/androidx/constraintlayout/widget/ConstraintLayout

[27] ConstraintLayout, https://developer.android.com/codelabs/constraint-layout

[28] ConstraintLayout, https://developer.android.com/guide/topics/providers/content-provider-basics

[29] Uniform Resource Identifiers (URI): Generic Syntax, https://www.ietf.org/rfc/rfc2396.txt

[30] Implementing ContentProvider MIME Types, https://developer.android.com/guide/topics/providers/content-provider-creating#MIMETypes

[31] Content Providers in Android with Example, https://www.geeksforgeeks.org/content-providers-in-android-with-example/

[32] Content Providers in Android with Example, https://developer.android.com/guide/topics/providers/content-providers

[33] Material Design 3, https://m3.material.io/

[34] Material Design, https://material.io/

[35] Material 3 Design Kit, https://www.figma.com/community/file/1035203688168086460

[36] Material Design color tool, https://material.io/resources/color/#!/?view.left=0&view.right=0

[37] Material Symbols and Icons, https://fonts.google.com/icons

[38] Gra Pokémon GO, Niantic, Inc., https://play.google.com/store/apps/details?id=com.nianticlabs.pokemongo

[39] Sensors Overview, https://developer.android.com/guide/topics/sensors/sensors_overview

[40] Request location permissions, https://developer.android.com/training/location/permissions

[41] Request location updates, https://developer.android.com/training/location/request-updates

[42] Request location updates, https://developer.android.com/training/permissions/requesting

[43] Permissions codelab Repository, https://github.com/android/permissions-samples/tree/main/RuntimePermissionsBasicKotlin

[44] Data and file storage overview, https://developer.android.com/training/data-storage

[45] Android's Room in Kotlin ft. MVVM Architecture and Coroutines, https://github.com/umangburman/MVVM-Room-Kotlin-Example/

[46] Kotlin coroutines on Android, https://developer.android.com/kotlin/coroutines

[47] Introduction to LiveData in Android, https://www.innominds.com/blog/introduction-to-livedata-in-android

[48] Retrofit – A type-safe HTTP client for Android and Java, https://square.github.io/retrofit/s

[49] Android Networking With Kotlin Tutorial: Getting Started, https://www.raywenderlich.com/6994782-android-networking-with-kotlin-tutorial-getting-started

[50] Android Networking With Kotlin Tutorial: Getting Started, https://github.com/julpanucci/Kotlin-Retrofit

[51] Composing suspending functions, https://kotlinlang.org/docs/composing-suspending-functions.html

[52] Representational state transfer, https://en.wikipedia.org/wiki/Representational_state_transfer

List of Figures

List of Tables

Listings

Mobile Application Development
Study material

Author: dr inż. Radosław Maciaszczyk
West Pomeranian University of Technology in Szczecin

Publisher: Mendel University in Brno, Zemědělská 1, 613 00 Brno, Czech Republic
Graphic editing and typesetting: Radosław Maciaszczyk with style by Jiří Rybička
Year of publishing: 2022
First edition
Number of pages: 80

ISBN 978-80-7509-890-0 (online ; pdf)
DOI: https://doi.org/10.11118/978-80-7509-890-0