Lappeenrannan teknillinen yliopisto

School of Business and Management

Sofware Development Skills

**Arto Savolainen, FITech-student**

**LEARNING DIARY, Software Development Skills: Mobile 2022-23**

**1.6.2023**

I registered my course enrollment in Moodle and read through the general course information. Started work on this diary. Checked out the environment setup tab - I already have VS Code set up with android emulator support (from playing around with Flutter development) and Git Desktop, so was free to skip this part. I then moved on to the Android mobile development tab. Change of plans - apparently these instruction videos use Android Studio, so I suppose I will do so as well. Luckily I already have it installed on my PC.

Downloaded and installed Java runtime and development kit. I've not written Java code since highschool so this should be an interesting course. Hopefully not a very frustrating one.

Created a new project in Android Studio and committed it. Watching the first video and I'm seeing a lot of XML files... not fond of that. Is this going to be a configuration nightmare?

**3.6.2023**

The layout editor is confusing. One can't just put down elements where one wants, they seem to have a life of their own. My first attempt at adding a simple text element below the toolbar in activity\_main has failed. The text element attaches itself on top of the toolbar and refuses to move. Not the start I wanted.

It appears the main activity view in this template is a CoordinatorLayout, which is a "super-powered FrameLayout" (obviously). Looks like this view contains the toolbar and space below it for content that is defined elsewhere. Apparently one is not supposed to just drag elements into such a layout. This template is not ideal for a beginner - there's too much going on here.

I started a fresh project from a simple "Hello world" template. Being prompted to extract hardcoded text into a string resource, I looked up Android naming conventions for files and resources. The Youtube video linked in the course material uses CamelCase for component IDs but googling suggest snake case might be the common way. I chose to use snake case for this project.

Finished the first instructional video and applied some of the techniques to my fledgling project. I now have a view with a couple of TextViews and an understanding of how to programmatically modify their contents. Learned the basics of debugging in Android Studio - it's the same as everywhere else.

I watched a couple more videos about Android development and figured out how to use Intents to start activities, i.e. transition between views in this case. I added a new Settings Views Activity into my project and created a button with a "settings" icon in MainActivity to navigate to it.

**4.6.2023**

Did more work on the settings screen of my app. Read the documentation and found out how to set an EditTextPreference to allow the user to input decimal numbers only. For an EditText this could simply be done with a single line of XML, but an EditTextPreference requires a more cumbersome approach through Java code. I wonder why that is? I could not find a reason for it, but I shan't assume laziness... hah.

Googled around and found that going back to the previous activity, i.e. popping the activity stack is very simple: just call finish(). Added a back button to SettingsActivity which calls finish() on click. Apparently this is called an "Up" button, and could be added to views automatically with an app bar. Considered configuring an app bar and a navigation system but decided against it. The app I'm developing only has three views, both of which are accessed from the main view, so the current method of navigation will suit my purposes perfectly.

I've spent most of this evening trying to figure out Android themes. Working with themes is absolutely terrible. The only way to edit a theme is to go through attributes in an XML file and give them values until you find the one you're looking for. And there are hundreds of attributes. You think iconTint might change the color of an icon in an ImageButton? No it doesn't, and nobody on the entire Internet knows what does. So in practice it is very time-consuming to define styles through themes, and many elements still had to be styled separately simply because I could not find the proper attribute. What I've learned tonight is that the Android theme system is trash, mostly because of scattered / non-existent documentation.

I examined the AndroidX Preference documentation and implemented some further validation for the EditTextPreference inputs on my Settings screen. I was expecting built-in methods for validating user input but the only thing I found was overriding the onPreferenceChange method and attaching that to the preference object. I will send an e-mail to the TA to ask about this problem as I would like to find a more elegant solution.

**5.6.2023**

Today I've been reading the Android developer guide's articles on background work. I need to figure out how to create a background worker that will periodically fetch data from a URL and persist even if the app instance is destroyed - this is to send notifications. For this purpose I've created a DataFetchWorker that is a subclass of Android's Worker class.

HttpUrlConnection looks cumbersome to use so I checked around for alternatives. My search brought me to <https://blog.codavel.com/android-http-libraries-landscape> which is a handy overview of HTTP query libraries for Android. I chose Volley for this since it looks simple enough and is recommended and maintained by Google.

Further examination - Volley works asynchronously which it makes it a bit more complicated to use within a Worker. OkHttp supports synchronous query execution so I will try that instead. A Worker runs in a background thread thus a blocking request is desired there. OkHttp docs had some syntax I'd not seen before - learned about the Java try-with-resources statement.

Preliminary implementation of a data-fetching Worker using OkHttp is functional. To check that the HTML it downloads is as expected I looked up how to write to a file on Android and extracted the .html file the app wrote with the Android Studio Device File Explorer. The HTML looks correct, I can continue with implementing the actual business logic.

Spent time thinking about state management. I started implementing my data store as a Singleton (actually there is no explicit instance that is returned, just static methods) - looks like this is a valid and common approach (edit from the future: there are in fact good reasons to oppose this tactic). I researched and tested how arrays, lists, and unmodifiableLists work in Java. When I used the List.of() method to create an unmodifiableList Android Studio suggested I upgrade the "language level" of the project to 9. Is it a bad idea to use Java 9 features? I tested this using a Nexus 6 emulator with an image of Android 7.1.1 targeting API 25. I changed my application's minSdk in build.gradle and voilà it works! I decided to test this further and downloaded Android 5.1 which is over 8 years old and the app works on it.

Thinking about it, it's no surprise that this particular method would work on older machines, since its functionality requires no changes in the bytecode or the VM itself, just the program compiling Java code into bytecode. But do Android apps run in a Java VM? This prompted me to investigate how Android actually works. Android apps do not in fact run in a standard Java virtual machine; Android versions previous to Lollipop used Dalvik VM, while today Android Runtime is used. ART takes the same bytecode format as Dalvik and compiles it into machine code upon installation. Newer Java APIs are supported on older API versions through a process called "desugaring", where the new libraries are included by the compiler with the rest of the bytecode. I think what happened is that changing the language level to 9 caused Android Studio to include Java 9 features in the app through desugaring. Does this have a significant effect on file sizes? It's late so that investigation will have to wait.

**6.6.2023**

When handling the response from an HTTP request I encountered a warning about potentially producing a NullPointerException. Did some reading about null handling in Java and it looks like using the ternary operator when possible is the least verbose way to implement null checks. Java has also introduced the Optional type, which I already have experience using in Rust. It might be a useful tool for a larger architecture but in this project I will stick with the more traditional method in the interest of time.

When logging HTML retrieved by DataFetchWorker with Log.d() I could not find the string I was looking for using Logcat's find function. Again I wrote the HTML to a file and was able to verify the downloaded data was valid. This enabled me to identify a bug related to parsing said data. It looks like Logcat cuts off lines that are too long - the file was 260k characters spread across only a few lines. Now I need to split a string using a delimiter but escaping characters in strings is inconsistent in Java and will throw up errors with perfectly logical escape sequences.

I spent a long time debugging an error that was actually the result of a bug in the editor / compiler. Editing the original delimiter string to add a sequence of escape characters erroneously produced compiler warnings and errors. I'm certain it wasn't a mistake on my part as I made sure the string literal was written correctly. Typing out the same sequence of characters and assigning it to another variable worked. Finally I'm done with this. Edit: I was not done with it. String.split() takes a regular expression as the delimiter, of course. Oh my god googling how Java regex works when this should be an extremely straightforward problem to solve in any reasonable language is aggravating.

Finally. It finally works. What did the trick was using Pattern.split() instead of String.split(). I found this method after over an hour of googling. So what could have been, in a better world, simply responseBody.split(delimiter) ended up looking like this: Pattern.compile(Pattern.quote(delimiter)).split(responseBody); What an absolutely terrible experience.

Finished work on DataFetchWorker. App state is now correctly updated whenever fetching completes. Next I will have to research how linking state to display components is typically done in Android.

I've been reading the Android docs and for updating views they recommend ViewModel and LiveData, where a LiveData object holds the data and the ViewModel manages it. I will try this approach.

The ViewModel for MainActivity is now working and the view is updated automatically when the MutableLiveData held by the view model changes. Now I need to figure out a way to update the MutableLiveData after UpdateWorker fetches new data. I think I will create a Mediator class to handle communication between StationsData (which holds app state) and the views.

I researched how to write and read SharedPreferences. I decided it was necessary to store the state of the currently selected station in SharedPreferences in case the app process is destroyed by the OS. Changed MainActivity's ViewModel to AndroidViewModel to be able to get context and thus retrieve data from SharedPreferences. Edit: As soon as I'd implemented this I realized it was not at all necessary to store state in SharedPreferences, and that data would be stale or at best redundant. All that's needed is storing the name of the currently selected station in SharedPreferences, to prevent the displayed station from changing every time the Activity or app process is destroyed.

Set API minSdk to 23 to be able to use TextView.setTextAppearance(), which changes the text style of a TextView. I also looked up how to prevent the app from switching to landscape mode, where bottom elements went off the screen. There's no point to landscape mode in this app, so I applied the necessary changes in AndroidManifest.xml

It took me a while to track down where my app failed. It didn't crash or anything, just didn't behave as excepted. I stepped through the code and found an uncaught exception from MutableLiveData: "Cannot invoke setValue on a background thread from Coroutine". I googled the error and saw that MutableLiveData has a method called postValue() which should be used instead of setValue() when calling from a background thread - the source of the call chain was a Worker. I made the fix and it worked! It was a great relief and I'm pretty surprised I could fix it by just using a different method. Points to the API for that. This is the first time data is actually updated on the screen properly.

I think app architecture is mostly in place now. It took me a while to get here. Even a small and relatively simple app gets rather complicated to implement because of lifecycles and the general structure of Android app. I've had to think about the architecture of the program a lot more than in previous projects where I've written an equivalent app using other frameworks. Thankfully the Android API offers enough tools to get the job done. I haven't had to worry about threads or asynchronous execution in any significant detail which is pretty great.

**7.6.2023**

Tested the app on older phones on the emulator and the results are confusing. The screens display and work as expected but with default values, the rest of the app just doesn't work at all. I'm going to download emulator images and see at which point the app starts working.

Pixel Nougat 7.1 - nope

Pixel 2 Orea 8.0 - no

Pixel 3 Pie 9.0 - same, none of the business logic works

Pixel 4a Android 11.0 - nada

I isolated the problem: UpdateWorker was not started by the WorkManager on Android versions older than 12. Apparently the .setExpedited() method of the work request builder only works on newer versions. Commenting out the call solved the issue - the app now works correctly on a Pixel API 25 Nougat 7.1 emulator, which is good enough for me.

New issue: on many Android versions pressing the backbutton and coming back to the app wipes state and the UI reverts to default values. It's weird that this happens on some API versions old or new, but not on for example a Nexus 6 running API 25. I'm going to have to investigate lifecycle management and figure out how to trigger refetch when the app resumes. Edit: looking into it, there are buit-in mechanisms for saving UI state which I should try to implement.

I also encountered some unpleasant flashing of the UI when transitioning between views on certain devices but I think and hope it's just an emulation issue. Devices where the emulation is smooth don't exhibit any such behaviour.

I read the docs for more information about lifecycles. I'm not sure if I should explore using services or continue with WorkManager. Edit: reading further it seems that JobIntentService, for instance, is deprecated in favour of WorkManager. I probably won't bother studying how services work then.

I remembered that Java has interfaces and implemented a StationDataSubscriber interface so DataMediator doesn't have to know about the type of object that subscribes to receive data updates.

I'm unsure if a regular background Worker is reliable enough to run continuously and not get killed by the OS. I looked at what a setForegroundAsync() does but it's for a different type of job. Despite looking all over the docs for a couple of hours I'm still unsure how to go about background work.

I read a helpful article (<https://www.kodeco.com/5212210-jetpack-saved-state-for-viewmodel-getting-started>)

Today I spent hours trying to comprehend lifecycles and the Saved State module. Do TextViews internally use SavedStateHandle or related functionality? Is there any point in implementing it in my ViewModel, or is practically already done automatically? It looks like a relatively new addition to the API - will it work on older phones? Is it already failing on older APIs and working on newer ones, and is this the reason for the inconsistent behaviour of my app across Android versions? I don't know the answer to any of these questions, some people on the internet say one thing and the docs make no mention of it, or it's buried somewhere so deep no one will ever find it. Trying to get a clear picture of how anything works in Android is borderline impossible. Just godawful. Makes me want to give up trying to do things properly.

I implemented SavedStateHandle in the MainActivity's view model. The implementation itself was simple. From testing on different phones it looks like it actually works. The phones that previously reverted the main view to default values when restoring from the background now persist view state. I assume some emulated combinations of hardware and Android are just much more aggressive in destroying activities than others.

**8.6.2023**

I've been thinking about a problem I have. WorkManager's inbuilt mechanism for periodically executing a Worker task has a minimum period of 15 minutes, but I want to update app state every 10 minutes, as that is how often the data source refreshes with new data. I will try to create a timer that queues a WorkRequest every 10 minutes in one of the singleton classes, WorkController probably, but I don't know how likely it is that this process will be killed, and that might be hard to test. Is the 15 minute cycle really the only reliable way to run a background task on a timer?

Ran into a problem using SavedStateHandle. ViewModel state was being updated by UpdateWorker running in a background thread, thus savedStateHandle.set() threw an exception. There's no thread-safe method for this. Because of uncertainty regarding lifecycle behaviour and process longevity I'd been planning a move to use SharedPreferences for holding state exclusively, and this forced me to start the refactoring.

Rewrote data update logic to store station data in SharedPreferences instead of StationsData. MainModel listens to changes with onSharedPreferenceChanged() and updates the LiveData which is observed by the view. I encoured a distressing problem where the key parameter was null and spent a couple of minutes panicking about it, thinking it had something to do with threadsafety. It turned out to be a redundant Editor#clear() that I accidentally left in the code. But that was just one of many bugs and I spent a few hours implementing this change.

I enabled strict mode to see if I could find out why Android was complaining about a resource failing to close. I couldn't - the trace that was provided was useless. There were so many warnings in the logs I decided I couldn't waste time trying to figure out which ones were relevant to me or actual problems that I should consider. But it did clue me in to the fact that libraries are grouped into SDK and non-SDK, and that using code from the latter may result in the app breaking. I checked <https://github.com/mercadolibre/mobile-dependencies_whitelist/blob/master/android-whitelist.json> and if that list is accurate then OkHttp is whitelisted, which makes sense.

I googled for information on how to automatically start an application on device startup. As far as I can gather, there's no way to do this natively on Android, only some phone manufacturers add the option. So the user has to manually start the app after every device reboot to start receiving notifications. Not great.

I ran the new state management system on emulators and it failed to work on older API versions. After some debugging I saw that onSharedPreferencesChange() was not triggering when preferences were written to. I googled the issue and found <https://stackoverflow.com/questions/3799038/onsharedpreferencechanged-not-fired-if-change-occurs-in-separate-activity> "This happen because garbage collector. its works only one time. then the reference is collected as garbage. so create instance field for listener." So I did and now it works on all models down to at least Nougat 7.1. And I don't really care about testing anything older than that.

Good information here: <https://developer.android.com/guide/components/processes-and-threads.html>. Don't block the UI thread! I think the current architecture of my app should avoid this for the most part. Requires testing.

**9.6.2023**

Today I spent some time reading the Android developer's guide. The picture is becoming clearer but I'm still unsure if a backgroud Worker runs in the Application process. I will have to setup adb on the emulator and kill the app to see what happens with the Worker and if it can still access StationData's methods.

Also read about Direct Boot mode and credential encrypted storage, not relevant to this project but interesting. Learned that Android applications share data via a ContentProvider, which offers methods for accessing the shared data. Requests are routed through a ContentResolver.

Read documentation about GridView and how to implement them. Again a simple thing requires way too much code. I had a problem with using the example code I copied from the documentation; setAdapter() did not accept the custom adapter type I'd created. Again the example in the docs was incomplete. I decided to ditch it and go with a different type of adapter following an example from another site.

Unexpected problem: I had a fatal error "java.lang.ClassCastException: com.example.avaruussaa\_android.InitApp cannot be cast to android.app.Activity" when trying to call finish() from within the adapter. InitApp is my Application class. After going through half a dozen Stack Overflow posts I found the answer: <https://stackoverflow.com/questions/33972621/cannot-use-finish-inside-adapter>. I'd erroneously given getApplicationContext() to the adapter instead of "this". This was the first time I had to pay attention to the source of context. Will need to be careful to pass proper context from now on.

Spent some time creating drawables and styling the app. Configuring the appearance of an Android app is infinitely more enjoyable than doing battle with CSS. Impressed by the ease of it, so far. Themes are still unoptimal though - Trying to find the right attributes by trial and error feels like a waste of time. Maybe there's a better way but I haven't found it. The setTextAppearance() function cannot override attributes set elsewhere so I had to style some elements individually. Not a big deal in a small app but in a larger project a solution for that would have to be figured out.

I decided to try the app on my phone. I enabled USB debugging, connected it to the PC and loaded the app. It looks pretty good and runs smooth. This is much nicer than using the emulator. I did try older versions on the emulator again and encountered an immediate crash: "<gradient> tag requires 'angle' attribute to be a multiple of 45". Kind of funny that this would crash the whole application when on newer APIs it very reasonably just works.

<https://developer.android.com/training/dependency-injection> recommended libraries for dependency injection in Android. Not useful for a small project but good to know for later. Hilt looks like it could be a good option.

**10.6.2023**

Why do I keep getting the warning "Expecting binder but got null!" every time I change views with startActivity()? I can't find an answer for this. It's not negatively affecting the app, as far as I can tell, but it's still bothering me.

I rewrote much of the state management code for improved performance and better maintainability. The speed doesn't really matter in practice but this way makes more sense. At first my app relied entirely on a static variable to hold data. Then, out of concern for the reliability of that approach, I changed everything to write to SharedPreferences instead. Now, data is both cached in memory and written to disk for safety. For a larger app I would look into using a DataStore but SharedPreferences works great here. Though I did read that DataStore is actually slower to use, so that's something to consider if performance is critical.

Working with Java has been pleasant when dealing with basic object-orientated programming. Class interaction has been smooth to implement. Try-catch is still cumbersome though. The main downside I've encountered so far is the tediousness of solving some very straightforward tasks - here Java is showing its age.

One of the main difficulties of developing for Android comes from the overall size and complexity of the Android API. It's huge and every time I look up how to do something I get three or five levels deep in the API documentation. It's a lot to take in. The other difficulty is dealing with the Android architecture which will kill and restart processes / activities as it pleases. Designing an app to work with this is a lot more effort than desktop development has been in my experience. And then there's the lack of convenience functionality in some areas. Why is user input validation left to the developer to implement manually in Java, for example? Surely the API could provide some help in this regard. For the most part the API has been sufficient, quite remarkably, as it has obviously been a colossal effort to develop it. I just wish the documentation was better organized. All API doc pages should ideally have examples and links to pages which describe the use of that particular class or feature - from what I've seen, most of them have nothing of the sort. Most time wasted so far has been trying to google how to do something and searching through years-old Stack Overflow posts because I just can't find what I need in the official documentation.

Android Studio / IntelliJ are great though. Code suggestions work very well and handling the interconnections of an Android app is easy. Emulator support is really nice. Has to be one of the best IDEs out there.