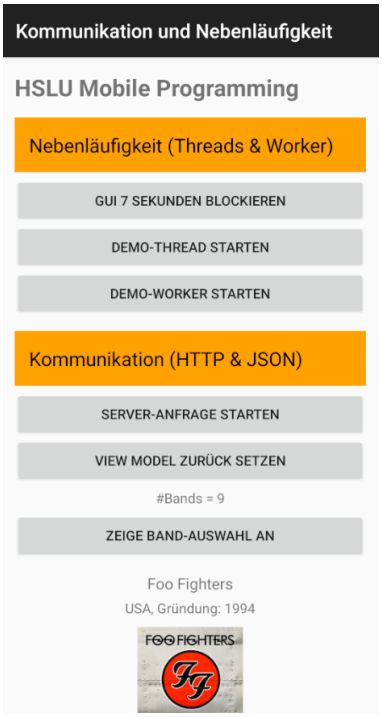


Übung 4: Kommunikation & Nebenläufigkeit – Threads, Workers, HTTP & JSON

In dieser Übung geht es um Nebenläufigkeit und Backend-Kommunikation unter Android. Konkret sollen länger dauernde Aufgaben nicht den main-Thread einer Android-Applikation blockieren. Sie implementieren dazu Threads und Workers. Weiter implementieren Sie HTTP-Anfragen, um JSON-Ressourcen mit Hilfe eines Retrofit-Services abzurufen und Sie stellen Bilder dar. Konsultieren Sie dazu wie gehabt die dazugehörigen Folien. Und falls Sie Fragen haben, stellen Sie diese bitte im Ilias Forum, bzw. schauen Sie, ob es dort mehr Infos zu dieser Übung gibt. ☺

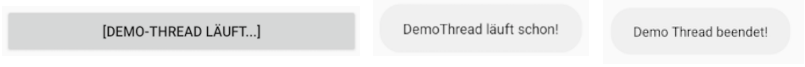
0. Neue App: Com & Con (Communication & Concurrency)

Erstellen Sie ein neues Android-Applikationsprojekt "Empty Activity". Diese Übung besteht aus verschiedenen Teilaufgaben, welche in der Main-Activity von diesem Projekt soweit als möglich resp. sinnvoll dargestellt werden. Diese `MainActivity` (Layout-Datei: `activity_main.xml`) wird am Schluss ungefähr so aussehen wie der Screenshot rechts oben, verwenden Sie dazu eine `ScrollView` mit einem `LinearLayout`, siehe Übung 2 für Details dazu. Hinweise: Verwenden Sie für diese Übung KEINE Fragemets. Und einen Grossteil vom Code dieser Übung implementieren am einfachsten direkt in der MainActivity. Und damit Sie diese Übung wie gewünscht implementieren können, müssen in der build.gradle-Datei von Ihrere App u.a. die folgenden Abhängigkeiten angegeben sein:

```
dependencies {
    …
    implementation 'androidx.appcompat:appcompat:1.2.0'
    implementation 'androidx.work:work-runtime-ktx:2.5.0'
    implementation 'androidx.activity:activity-ktx:1.2.1'
    implementation 'com.squareup.retrofit2:retrofit:2.5.0'
    implementation 'com.squareup.retrofit2:converter-moshi:2.4.0'
    implementation 'com.squareup.picasso:picasso:2.71828'
```

1. Blockier-Knopf

Fügen Sie der `MainActivity` einen neuen Knopf "GUI 7 Sekunden blockieren" hinzu. Wird dieser gedrückt, soll der aktuelle Thread (= main-Thread = UI-Thread) 7 Sekunden warten mittels `Thread.sleep(...)`. Entsprechend ist die App 7 Sekunden lang blockiert und Sie reagiert nicht auf Benutzerinteraktion... ☹



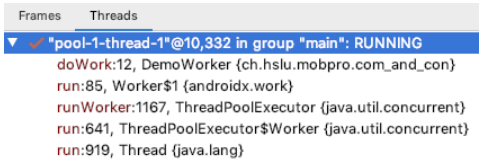
2. Warten in eigenem Thread

Fügen Sie nun der `MainActivity` einen neuen Knopf "Demo-Worker starten" hinzu. Wird dieser gedrückt, soll in einem eigenen Thread 7 Sekunden gewartet werden mittels `Thread.sleep(...)`. Während der Ausführung des Threads soll der Text des Buttons "[DemoThread läuft…]" lauten und bei erneutem Drücken soll ein Toast mit " DemoThread läuft schon!" erscheinen. Wenn der Thread beendet ist, soll ein Toast "Demo Thread beendet" erscheinen. Implementieren

Sie dazu einen eigenen `Thread` mit dem Namen " hsluDemoThread" und schauen Sie, dass von diesem Thread zu jedem Zeitpunkt maximal einer am Laufen ist. Setzen Sie in Debug-Breakpoints um die Ausführung in verschiedenen Threads nachvollziehen zu können, siehe Screenshot rechts.

3. Warten im eigenen Worker

Wird der Knopf "Demo-Worker starten" gedrückt, soll in einem Worker 7 Sekunden gewartet werden mittels `Thread.sleep(...)`. Implementieren Sie dazu eine eigene Klasse `DemoWorker`, welche von `androidx.work.Worker` erbt. Die Wartezeit soll dem DemoWorker mit Hilfe von dessen vorgesehen Mechanismus mittels `WorkerParameters.inputParameters` übergeben werden. Verwenden Sie zum Erzeugen Ihrer `DemoWorker`-Instanz einen `OneTimeWorkRequestBuilder` und setzen in diesem also mittels der Builder-Methode `setInputData` das gewünschte `Data`-Objekt (mit den 7000ms als `Long`). Setzen Sie auch hier wieder Breakpoints, um nachzuvollziehen, auf welchem Thread nun gewartet wird, siehe Screenshot rechts.



4. HTTP-Anfrage: JSON abholen inkl. ViewModel

Nun soll die App von folgender Schnittstelle Daten holen und verwenden:

```
https://wherever.ch/hslu/rock-bands/all.json
```

Die in dieser JSON-Datei vorhandenen Daten sollen mittels der Bibliothek *Retrofit* mit HTTP-GET geholt und mittels der Bibliothek *Moshi* in Kotlin-Objekte der folgenden Klasse konvertiert werden:

```
data class BandCode(
    val name: String,
    val code: String
)
```

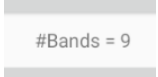
Ausgelöst wird diese Anfrage durch den Knopf "Server-Anfrage starten". Die durch diese Anfrage erhaltene `List<BandCode>` soll in einem `ViewModel` gehalten werden. Implementieren Sie dazu die folgende Klasse mit Property `bands`:

```
class BandsViewModel : ViewModel() {
    val bands: MutableLiveData<List<BandCode>> = MutableLiveData()
    …
```

Damit dieses in der MainActivity verwenden werden, deklarieren und initialisieren Sie in der `MainActivity`-Klasse ein entsprechendes Property. Das geht mit den Kotlin-Extensions für Android kompakt mittels (und ohne explizite Verwendung von einem `ViewModelProvider`):

```
private val bandsViewModel: BandsViewModel by viewModels()
```

Setzen Sie mittels `bandsViewModel.bands.observe(…)` einen `Observer`, damit von diesen Daten auf der Bildschirm der MainActivity etwas sichtbar wird. Verwenden Sie dazu in `activity_main.xml` eine `TextView` mit ID "main_nubmer_of_bands", auf welcher (wie im Screenshot rechts) die Anzahl der aktuellen vorhandenen Bands angezeigt wird.

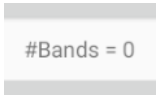


5. ViewModel zurücksetzen

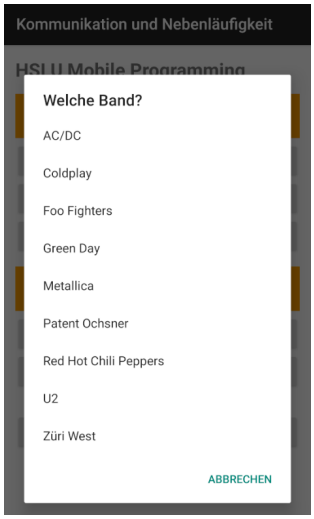
Durch Drücken vom Knopf "View-Model zurücksetzen", sollen allfällig geladene Bands zurückgesetzt werden. Implementieren Sie dazu in der Klasse X die folgende Methode:

```
fun resetBandsData()
```

In dieser Methode soll dem Property `value` vom `MutableLiveData`-Objekt `bands` eine leere Liste zugewiesen werden, das geht in Kotlin kompakt mittels `bands.value = emptyList()`. Nach Aufruf diese Methode soll die Anzeige auf der MainActivity anzeigen "#Bands = 0".



6. Band-Auswahl & Detail-Infos inkl. Bild



Zum Abschluss sollen die vorhandenen Daten auswählbar und angezeigt werden. Beim Drücken von "Zeige Band-Auswahl an" soll ein Dialog wie rechts im Bild mit allen vom Server geholten Bands (Property `BandsViewModel.bands`) angezeigt werden. (Infos zur Verwendung von Dialogen finden Sie im Foliensatz zu Android-2 bzw. in der dazugehörigen Übung.) Wird in diesem Dialog eine Band ausgewählt, sollen zu dieser vom Server weitere Infos geholt werden unter folgender URL (wobei [CODE] für den entsprechenden Band-Code aus `BandCode` steht):

```
https://wherever.ch/hslu/rock-bands/info/[CODE].json
```

Erweitern Sie Ihren-Retrofit-Code entsprechend, so dass Sie auf diesen Pfad die passenden Anfragen stellen können. Und als Base-URL können Sie also `https://wherever.ch/hslu/rock-bands/` verwenden. Und die so geholten JSON-Daten sollen analog zu den BandCode-Daten in Instanzen von folgender Klasse geparst werden:

```
data class BandInfo(
        val name: String,
        val members: List<String>,
        val foundingYear: Int,
        val homeCountry: String,
        val bestOfCdCoverImageUrl: String?
)
```

Erweitern Sie dazu u.a. die Klasse `BandsViewModel` um folgendes Property für die aktuell anzuzeigende Band (bzw. `null` falls aktuell keine Band angezeigt wird):

```
val currentBand: MutableLiveData<BandInfo?>
```

Und schauen Sie, dass in der `MainActivity` ebenfalls ein entsprechender `Observer` auf dieses Property registriert ist und dieser das GUI wie gewünscht für die ausgewählte Band anpasst, siehe Screenshot rechts unten. Das Band-Bild können Sie direkt in der `MainActivity` wie auf den Folien gezeigt mittels der Bibliothek Picasso von der URL holen und auf dem GUI setzen. Stellen Sie sicher, dass keine Band-Infos angezeigt werden, wenn das ViewModel zurückgesetzt wurde. (Auf `TextView`s können Sie z.B. einfach leere Strings setzen und eine `ImageView` lässt sich mittels `View`-Property `visibility` auf `GONE` bzw. `VISIBLE` setzen. ☺