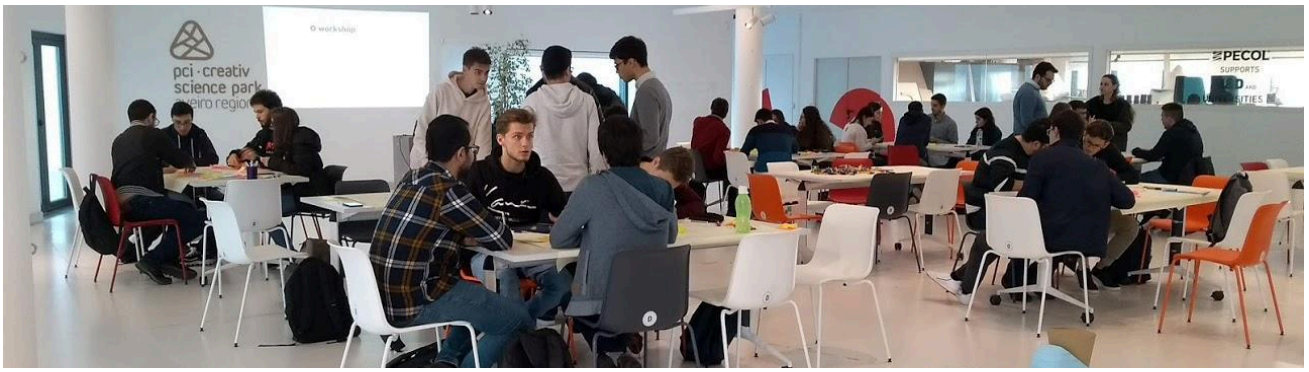
UA/DETI – Computação Móvel
Ilídio C Oliveira (ico@ua.pt)

Last revision: v2024-11-05 ***work in progress***



CM Weekly Activities – Android module

Introduction

Classes plan

See the overall course schedule, in Moodle.

Learning resources

Selected Android learning resources:

- Official Android developers' documentation: tutorials, API documentation, tools, best practices,...
- Main resources for CM classes:
  - "Android Basics with Compose: course"
  - "Android Development with Kotlin: Classroom course" (with lecture slides)
- Other Android training resources:
  - Kotlin course by JetBrains
  - Android Courses by Google with codelabs.
  - YouTube: Android Developers channel; Philipp Lackner channel.
- Books
  - Access these books at O'Reilly → CM playlist
    - "Programming Android with Kotlin" (2021) by P. Lawrence et al. → [PAwK]
    - "Android Programming: The Big Nerd Ranch Guide, 5th Edition" (2022)
    - "Professional Android" (2018) by Reto Meier

Week #1- Introduction to Android development workflow and tools

A) Kotlin basics

Concepts & readings:

Kotlin concepts:

- Slides from lesson 1 Kotlin Basics → Lesson 2 Functions → Lesson 3 Classes and Objects
- Kotlin essentials (for developers): PAwK→chap #1

See also:

- Kotlin style guide
- Kotlin online playground (run code in the browser)

Hands-on exercises (with code labs):

- Take the codelabs for Unit #1 - Introduction to Kotlin as needed. [suggested: activity #6]

B) Introduction to Android native apps development

Android fundamentals :

- The Android stack
- Build UI with (Jetpack) Compose (short tutorial)

See also:

- API levels

Hands-on exercises & code labs

- Unit 1 > "Setup Android Studio" [if needed]. Includes: "how to connect you [physical] Android device".
- Unit 1 > "Build a basic layout" [activities #3 and #4. Suggested for homework → #5]
- Unit 2 > "Kotlin fundamentals" [activities #3, #4 and #5]
- Unit 2 > "Add a button to an app" [activities #2 and #3. Suggested for homework → #4]

Week #2: Composable and UI state

C) UI, events and state managemnt

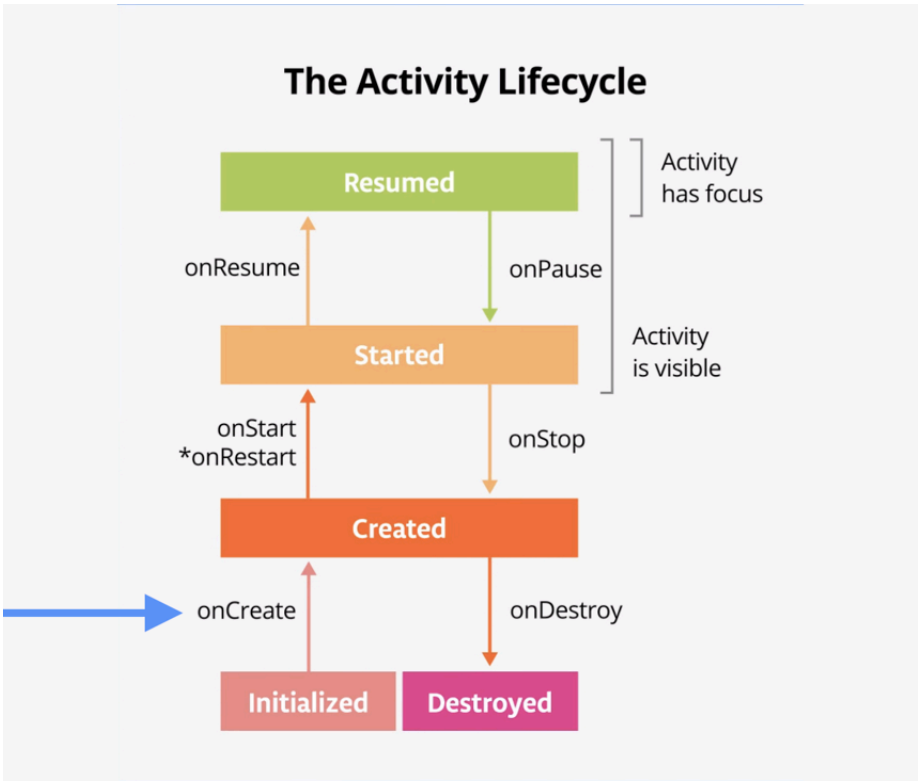
Resources and readings:

- Thinking in Compose: theory and principles
- Guide to Android App architecture
- Unidirectional data flow (UDF) pattern

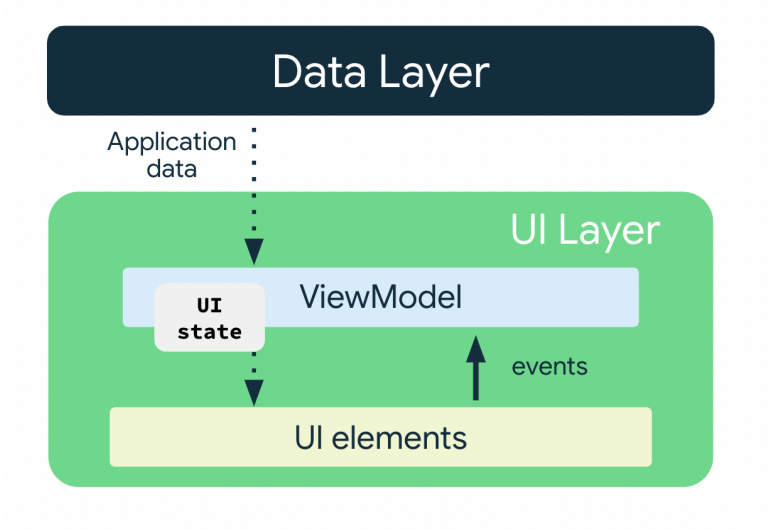
Hands-on exercises & code labs

- ☞ [Optional] Hands-on: Unit 2 > Interacting with UI and State: Tip calculator [Activities → #3, #4]. Focus on "state hoisting"
- ☞ Hands-on: Unit 4 > "Architeture Components" [Activities → #2, #5] Focus: UI state and ViewModel.
- ☞ [Optional] Scrollable list: Unit 3 > "Build a scrollable list" [Activities → #2].

Activity lifecycle



UI State and software design patterns



The UI layer is made up of the following components:

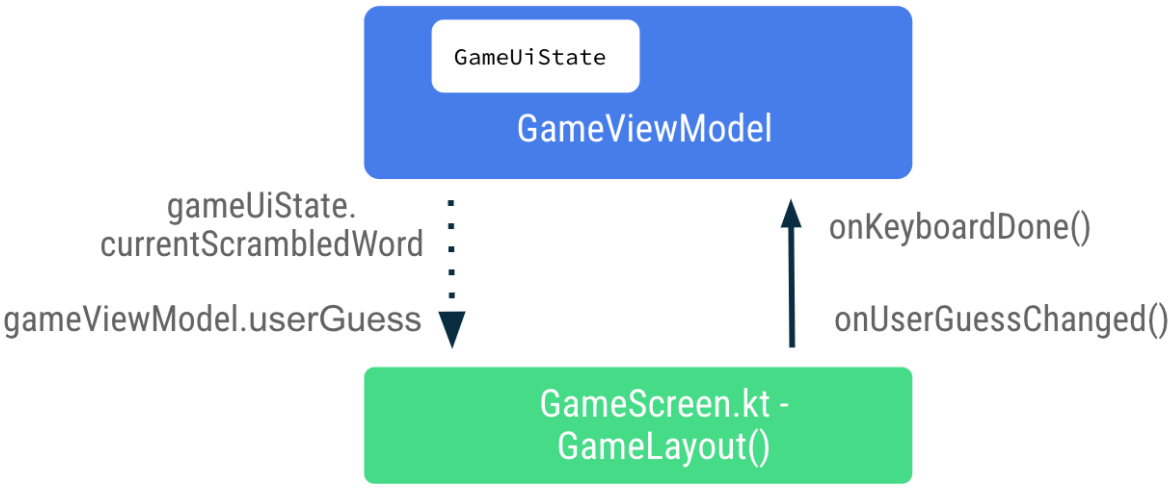
- UI elements: components that render the data on the screen. You build these elements using Jetpack Compose.
- State holders: components that hold the data, expose it to the UI, and handle the app logic. An example state holder is ViewModel. The ViewModel component holds and exposes the state the UI consumes. The UI state is application data transformed by ViewModel. ViewModel lets your app follow the architecture principle of driving the UI from the model.

Unidirectional data flow pattern

A unidirectional data flow (UDF) is a design pattern in which state flows down and events flow up. By following unidirectional data flow, you can decouple composables that display state in the UI from the

parts of your app that store and change state.
The UI update loop for an app using unidirectional data flow looks like the following:

- Event: Part of the UI generates an event and passes it upward—such as a button click passed to the ViewModel to handle—or an event that is passed from other layers of your app, such as an indication that the user session has expired.
- Update state: An event handler might change the state.
- Display state: The state holder passes down the state, and the UI displays it.



Homework 1: my Watch list

- **study this** example (it is a hands-on with code-along video support).
- create a simple application to manage your streaming "Watch list" (for movies and series).
- you should be able to add entries to the Watch List (stuff you want to watch) and, later, you should be able to mark entries as already watched (e.g.: tick a box). For convenience, start the app with a few entries pre-filled. Allow deletions too.
- you should use Composables, stateless UI, a list and the ViewModel pattern. Try to rotate the device while using the app (the list should not be lost with configuration changes...).
- for now, it is not needed to save the information in a persistent way (i.e., save to a database), but, if you want to, conside using Preferences Datastore.
- update your personal git with the Homework 1 implementation.

Week #3: Connecting to remote resources

Project resources:

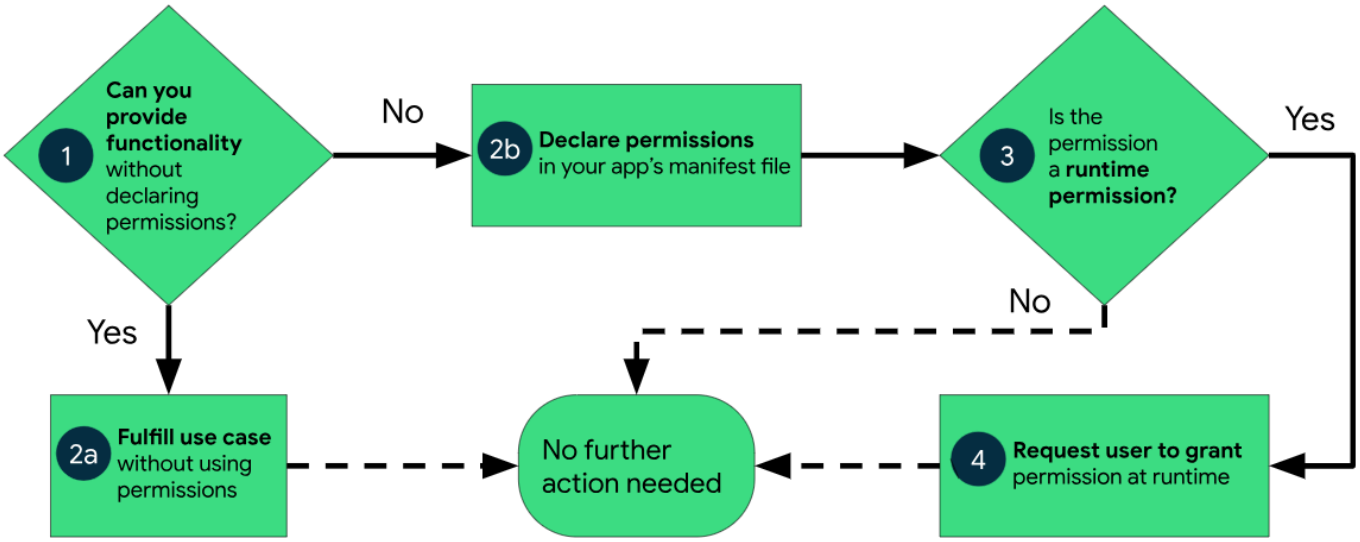
- Custommer/user journey map essentials
- Sample app project, full code, also deployed in Play store: "Now in Android"

D) Accessing (REST) data from the internet

Certain operations can be slow or affected by latency, negatively influencing a fluid user experience. "Heavy" or "slow" tasks should occur outside the default "main thread". Kotlin provides a convenient programming abstraction, the coroutine, to execute asynchronous code in "suspendable" functions. Getting data from the internet is an example of a programming case that can benefit from coroutines.

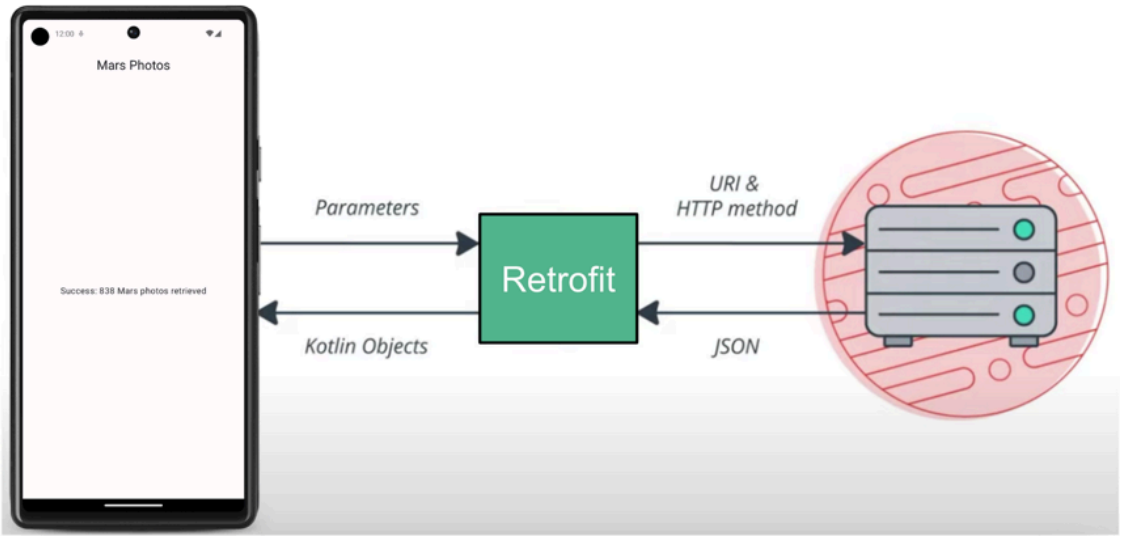
Resources and Readings:

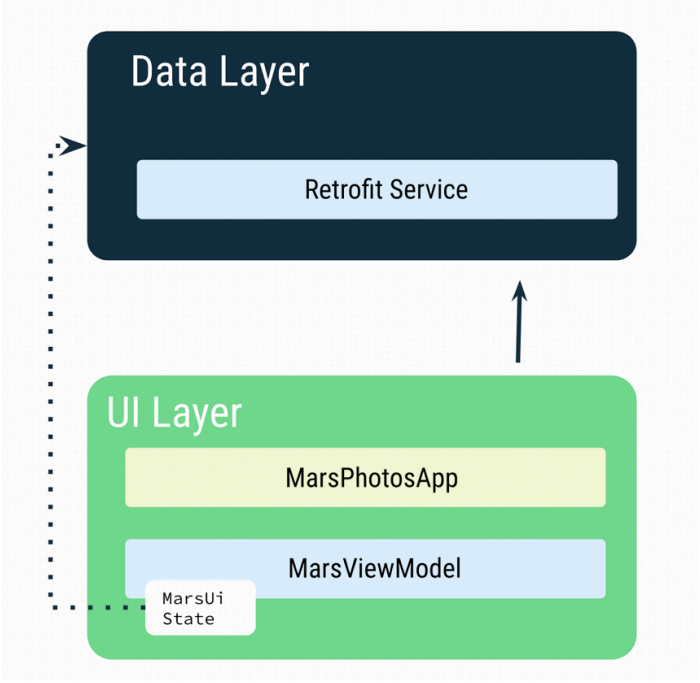
- Slides for coroutines (lesson 9, slide 29+)
- Slides for "Connect to the internet" (lesson 11)
- Requesting "permissions" guidelines. Pattern to request runtime permissions dynamically.



Hands-on exercises & code labs

☞ Hands-on: Unit 5 > Get data from the internet > Activities → #2, #5. Focus on: coroutines and non-blocking programming.





E) Mobile Backend-as-a-Service (MBaaS) integration - Firebase

Readings & Resources:

- Firebase services
- Starting with Friebase in Android: setup an project to use Firebase.
- Understand Firestore data model. Cloud Firestore is a NoSQL, document-oriented database. Unlike a SQL database, there are no tables or rows. Instead, you store data in documents, which are organized into collections.

Frequent Firebase use cases for mobile applications:

- Set up a user authentication flow with Authentication.
- Create a central database with Cloud Firestore (generally preferred) or Realtime Database.
- Store files (media), like photos and videos, with Cloud Storage.
- Trigger backend code that runs in a secure environment with Cloud Functions.
- Send notifications with Cloud Messaging.
- Gain insights on user behavior with Google Analytics.

Hands-on exercises & code labs:

- ☞ Hands-on: complete Build Friendly Chat code lab. (shows how to configure the Android project and interact with the Firebase backend.)

Week #4: Data access and location sensing

F) Data layer using ROOM and recommended architecture

Readings & resources:

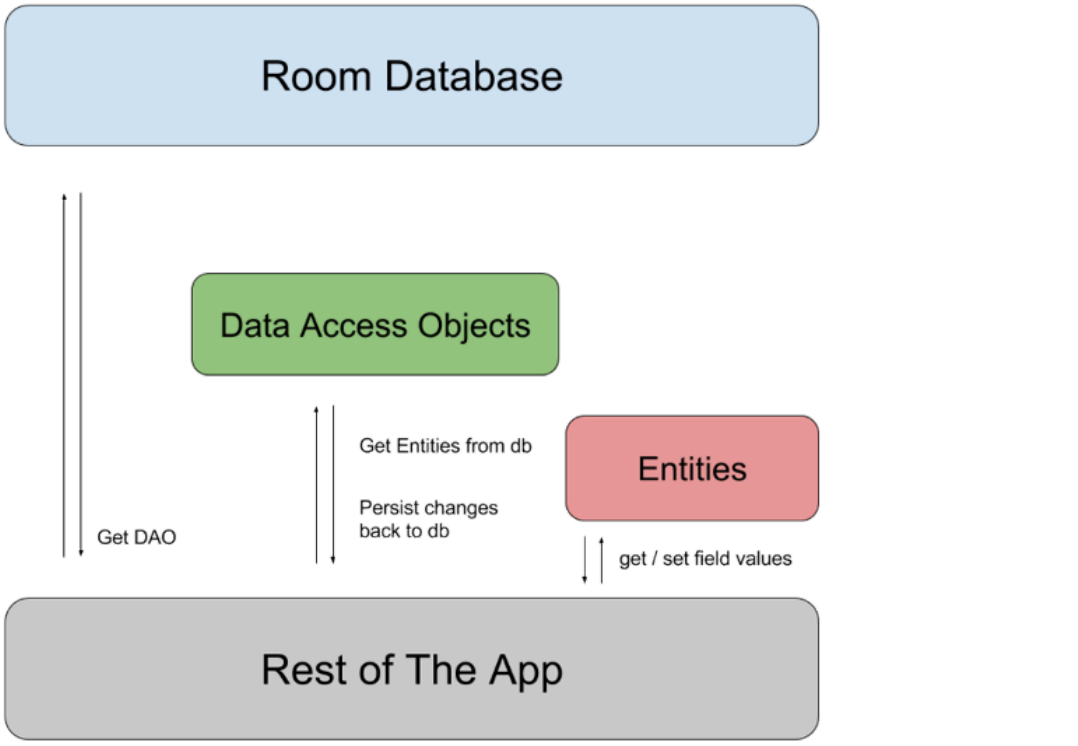
- Android architecture: the data layer.
- Slides for Room and app architecture (Persistence, lesson 9). Accessing data with the

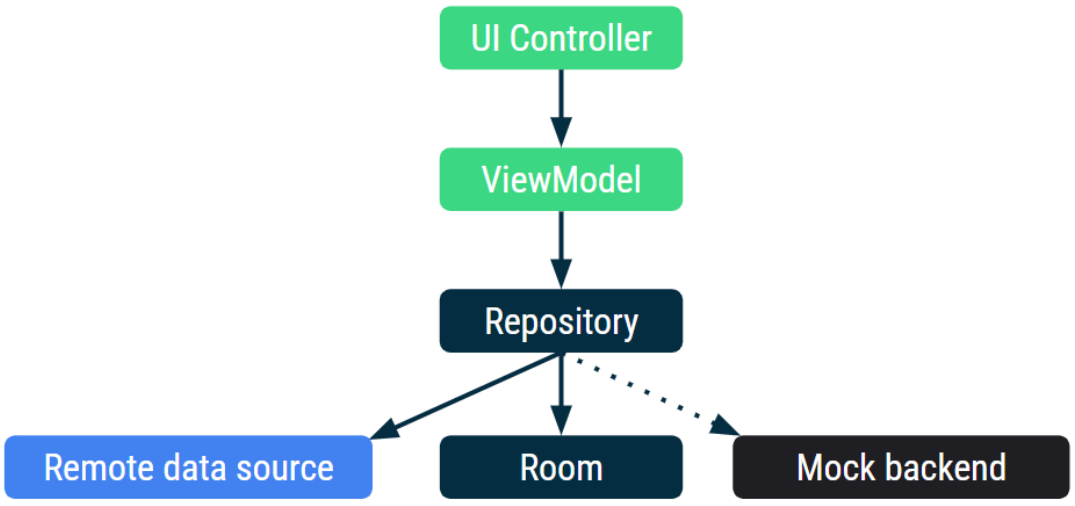
Repository pattern: book section & slides (lesson 12, initial slides only)

- Simple key-value pair storage: Preferences DataStore. (does not require a predefined schema, and it does not provide type safety or referencial integrity).
- More info on local databases/Room: book chapter → AP-BNRG, chap. 12

Hands-on exercises & code labs:

☞ Hands-on: Unit 6 → Get Use Room for data persistence (activities #4 Persist Data with Room, #5 Read and update data with Room)





G) Sensors and location updates

Readings & resources:

- Overview of the "Location and Context API" services, by Google. Requires Google Play Services on the target devices.

- Sensors API: concepts and programming model to access built-in Sensors.
- Concepts, API, and best practices to build location-aware applications
- Documentation on the Service application component (specially: foreground Services).
- Tracking and saving users location: video 1; video 2.
- See also: Android platform samples (location uses cases,...)

Hands-on exercises & code labs:

- Code lab: complete the "Receive location updates in Kotlin"
- [Suggested] Add a (Google) map to your Android app

Week #5: Deferrable work, DI & edge AI

H) The WorkManager API

Readings & resources:

WorkManager is an Android library that runs deferrable background work when the specified work constraints are satisfied. WorkManager is intended for tasks that require a guarantee that the system will run them (remains scheduled through app restarts and system reboots). WorkManager is the primary recommended API for background processing. Supersseeds other previous API (Job Scheduler,...). WorkManager supports chaining (e.g.: a pipeline of tasks to execute in sequence) and flexible scheduling constraints (e.g.: "run only when WiFi present").

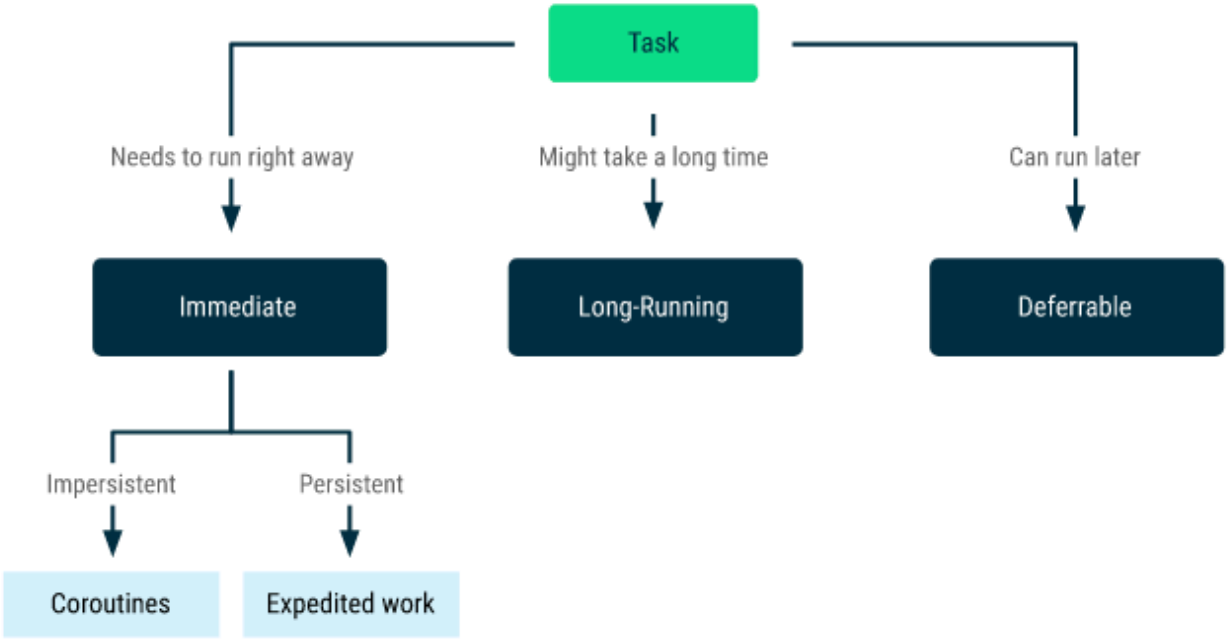
- Slides for lesson #12 (slide 10+)
- Guide to "Schedule tasks with WorkManager". Another guide (blog post).
- Background tasks overview: reacting to user actions and events

Some examples of tasks that are a good use of WorkManager:

- Periodically querying for latest news stories.
- Applying filters to an image and then saving the image.
- Periodically syncing local data with the network.

Hands-on exercises & code labs:

☞ Unit 7 → Schedule tasks with WorkManager: Activity #3 - Background work with WorkManager



Use Case	Examples	Solution
Guaranteed execution of deferrable work	• Upload logs to your server • Encrypt/Decrypt content to upload/download	WorkManager
A task initiated in response to an external event	• Syncing new online content like email	FCM + WorkManager
Continue user-initiated work that needs to run immediately even if the user leaves the app	• Music player • Tracking activity • Transit navigation	Foreground Service
Trigger actions that involve user interactions, like notifications at an exact time.	• Alarm clock • Medicine reminder • Notification about a TV show that is about to start	AlarmManager

I) Dependency injection

Resources & Readings:

Dependency injection refers to the programming pattern of providing (i.e., injecting) dependencies into a class rather than having the class instantiate or look for its required dependencies. Dependency injection provides yo the following advantages:

- Reusability of classes and decoupling of dependencies: It's easier to swap out implementations

of a dependency.
- Ease of refactoring: The dependencies become a verifiable part of the API surface, so they can be checked at object-creation time or at compile time rather than being hidden as implementation details.
- Ease of testing: A class doesn't manage its dependencies, so when you're testing it, you can pass in different implementations to test all of your different cases.

See also: Dependency injection in Android

Hands-on exercise:

☞ Codelab → Using Hilt in your Android app

J) Extending the app with AI

You can extend your mobile application with AI. A common use case is taking advantage of the device camera for image recognition/classification.

Google offers the ML Kit mobile SDK that brings Google's machine learning expertise to Android and iOS apps. Includes **Vision + Natural Language** APIs to solve common challenges in your apps, using models available from Google. The ML Kit APIs run on-device, allowing for real-time use cases (some updates may depend on the cloud).

For some use cases, instead of the generic ready-to-use cloud-models, you may want to develop your own machine learning model for edge devices, such as running in Android:

- The MediaPipe Solutions framework offers extensive support for image procession.
- The LiteRT framework (previously: TensorFlow Lite) makes it easy to apply TensorFlow models in your app.

Hands-on exercises & code labs:

☞ CodeLab: Handwrite digit classifier using MediaPipe Tasks

Project Topics - Android module

The Android group project assignment includes building a functional application that is (almost) ready for end users. You should implement mobility-specific use cases using Android programming best practices according to the criteria below.

Use of AI and other generative tools

Acceptable use of generative AI tools: It is natural to use intelligent assistants that increase productivity, such as the "copilots" built into Android Studio (e.g.: Gemini Assistant, GitHub Copilot). However, you are expected to design the application structure, justify architectural decisions, and be able to explain every element of your code. Full delegation of the "engineering" process, such as using

a declarative interface to generate a complete Android application, is not acceptable (e.g.: Cursor AI editor), or "low-code" environments (e.g.: OutSystems mobile app development).

Assessment criteria

Assessment criteria	Main Topics
A) Goals achievement: mobility use cases	Addresses use cases suitable for mobile/mobile first. Camera-intensive, geo (maps and/or location-intensive). Proactive notifications (local and/or Push notifications) Sensors (built-in or connected sensors) and adaptation to (sensed) user context UX coherent with mobility use cases and language (e.g.: content first,...) Others depending on the nature of the project
B) Complexity (of the implementation)	Adopts architeture guidelines/patterns and best practices Implements a backend strategy Data layer (strategy for); still useful if offline User-readiness (is UX ready for users?) UX continues across devices Others depending on the nature of the project
C) Effort	Fair effort for the team (taking into consideration the starting baseline for each student profile)

Seed projects (if needed): proposal.