



Lab on apps development for tablets, smartphones and smartwatches

Week 7: Coroutines, Room and Maps

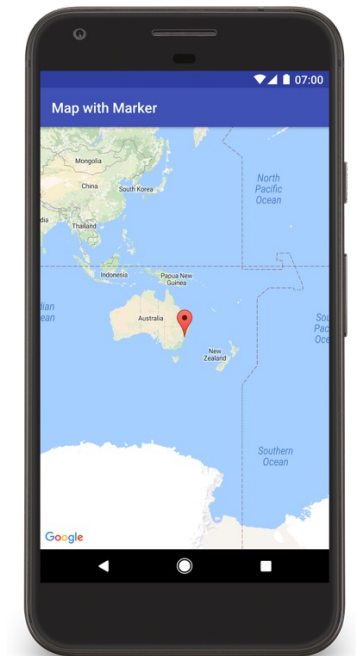
Giovanni Ansaloni

Dimitra Tatli, Riselda Kodra, Yuxuan Wang
Qunyou Liu, Amirhossein Shahbazinia, Christodoulos Kechris

School of Engineering (STI) – Institute of Electrical and Micro Engineering (IEM)

Class outline

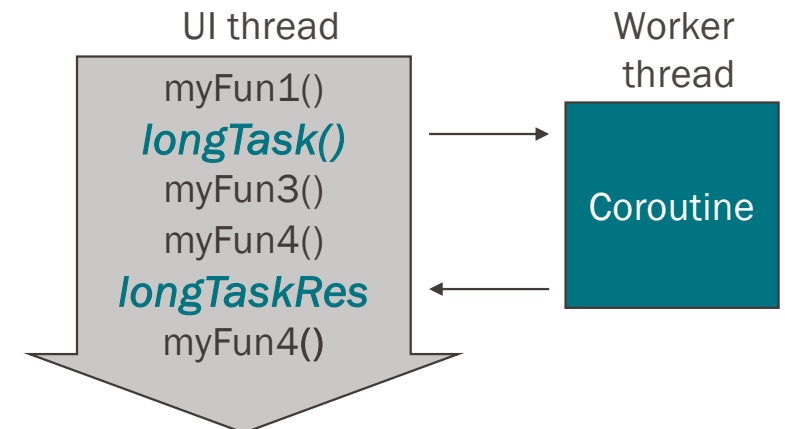
- **Coroutines and Room**
- **Geolocation**
 - GoogleMaps API
 - Location system service



- The UI must be always fast:
 - Screen is updated every 16ms → UI thread has 16ms to do all the work



- Coroutines are (long-running) tasks on a background thread
 - non-blocking
 - asynchronous





Coroutines in Kotlin

- Functions that can be invoked as coroutines are marked with the **suspend** keyword → `suspend fun longrunningWork() {...}`
- Every coroutines has associated
 - a **Job**: a handle to the coroutines
 - a **Dispatcher**: mechanism to send coroutines to different threads
 - `Dispatcher.IO` → I/O tasks
 - `Dispatcher.Default` → CPU-intensive tasks
 - `Dispatcher.Main` → Main thread
 - a **Scope**: context in which the coroutine runs
 - `ViewModelScope` → coroutines are destroyed if `ViewModel` is cleared
 - `LifecycleScope` → coroutines are destroyed if Lifecycle owner (`Activity`) is cleared



Coroutines in Kotlin

- Functions that can be invoked as coroutines are marked with the **suspend** keyword → `suspend fun longrunningWork() {...}`
 - a **Job**
 - a **Dispatcher**
 - a **Scope**: context in which the coroutine runs
In composables, `rememberCoroutineScope()` returns the composable scope

```
@Composable
fun myComposable(){
    val coroutineScope = rememberCoroutineScope()

    Button(
        onClick = {
            coroutineScope.launch {
            }} {...}
    )
}
```

From previous Lecture!

Launching a coroutine

- A coroutine is launched in a scope, specifies a dispatcher

```

fun someWorkNeedsToBeDone() {
    val job : Job = viewModelScope.launch {
        suspendFunction()
    }
}

```

scope →

```

suspend fun suspendFunction() {
    withContext(Dispatchers.IO) {
        longrunningWork()
    }
}

```

Coroutine →

Dispatcher →

- Suspended functions do not block execution while waiting for results
 - other useful work can be done
 - e.g. update GUI, listen for user actions...



- Most apps need data to be saved
 - persistent even when user closes the app
- Room provides that functionality via Room, an abstraction layer over **SQLite**
 - simplifies setting up and interacting over SQL databases
 - provides a query syntax based on SQL
- Apps interact with the database using normal function calls

Room library





SQLite and Room

- SQLite data in tables of rows and columns (spreadsheet...)
 - Field := intersection of a row and column
 - Rows are identified by unique IDs
 - Column names are unique per table

- Room links the Kotlin and the SQL syntaxes

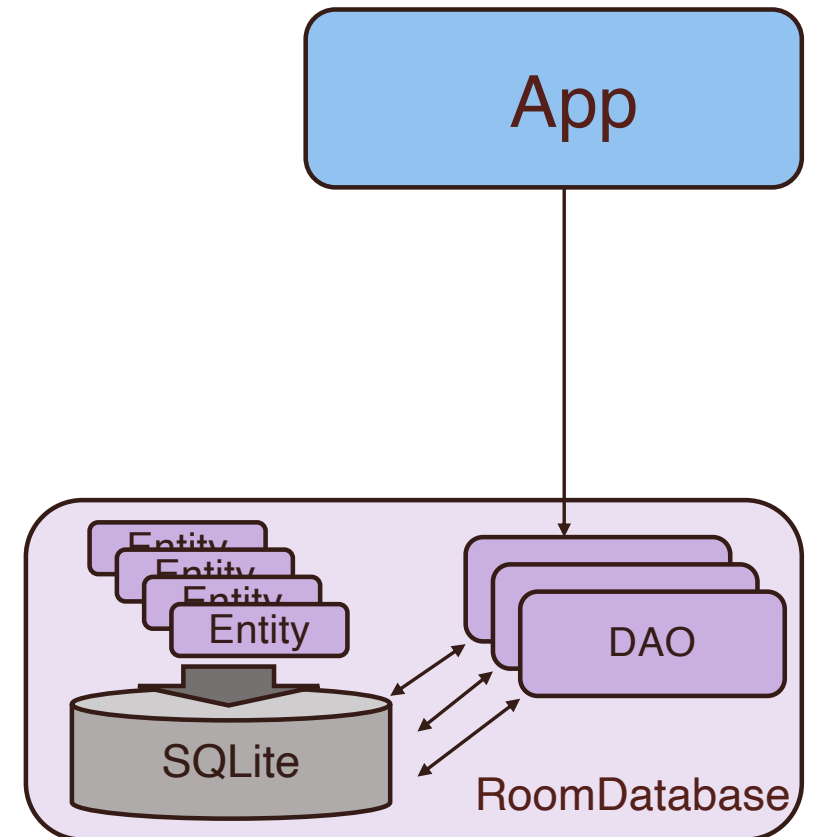
SQL → `@Query("SELECT * from my_table WHERE myId = :key")`
Kotlin → `suspend fun get(key: Long): myTableRow?`



- Three major components

- **Database**
main access point to DB
- **Entity**
class: describes a table within the database
object: one table row
- **Data Access Objects (DAO)**
Functions for accessing the database

Room library





Room Entity

- Kotlin data class with `@Entity` annotation

- optional `tableName` annotation
- unique `@PrimaryKey` field
 - can be auto-generated
- other fields
 - optional `@ColumnInfo` annotation

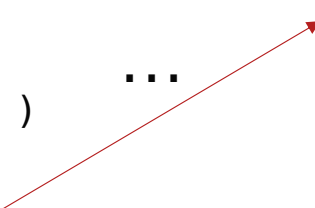
```
@Entity(tableName = "my_table")
data class MyEntity(
    @PrimaryKey(autoGenerate = true)
    var myId: Long = 0L,

    @ColumnInfo(name = "a_string")
    val aString: String = "",

    @ColumnInfo(name = "a_Int")
    var aInt: Int = -1
)
```

- Annotations are used to construct queries in the DAO (next slide)

```
@Query("SELECT * from my_table WHERE a_int = :intParam")
suspend fun get(intParam: Int): List<myTableRow?>
```





Room Data Access Object (DAO)

- Room databases are accessed by the app (e.g. ViewModels) using methods defined in DAOs
- DAOs provide mapping between Kotlin methods and SQL queries
 - DAOs are interfaces → the implementation of methods is generated by Room based on SQL code

```
@Dao  
interface MyDatabaseDao {
```

@Insert, @Update, @Delete

*→ convenience methods,
do not require any extra code*

```
    @Insert  
    fun insert(myTableRow: MyEntity)
```

```
    @Query("SELECT * from my_table WHERE myId = :key")  
    fun get(key: Long): MyEntity?
```

*Arbitrary queries
are defined with @Query*

```
}
```



Room Database

- Class annotated with `@Database`
- Only one instance needed for the app → Singleton
- `getInstance()` to either grab a handler of existing database, or create one

```
@Database(entities = [MyEntity::class], ...)
abstract class MyDatabase : RoomDatabase() {

    abstract val myDatabaseDaoInstance: MyDatabaseDao

    companion object {
        fun getInstance(context: Context): MyDatabase {
            ...
        }
    }
}
```

Entities used by the database

DAOs used by the database

getInstance() method in companion object



Performing Room queries

- Ultimately, databases should be accessed

- get an handler to the DB instance

```
val dataSource = MyDatabase.getInstance(application).myDatabaseDaoInstance
```

- We can now access the DAO methods

```
class MyViewModel(  
    val databaseDao: MyDatabaseDao,  
    application: Application) : AndroidViewModel(application) {
```

→ *DAO*

...

```
private fun fun1(key: Long): MyEntity? {  
    return databaseDao.get(key)  
}
```

→ *DAO
method
(query)*



Room and coroutines

- Accessing database can be slow → delegate it to coroutines!

1. mark DAO methods as *suspend*

```
@Dao
interface MyDatabaseDao {
    @Query("SELECT * from my_table WHERE myId = :key")
    suspend fun get(key: Long): MyEntity?
}
```

2. Call the DAO method in a co-routine
 - Room automatically uses the I/O dispatcher

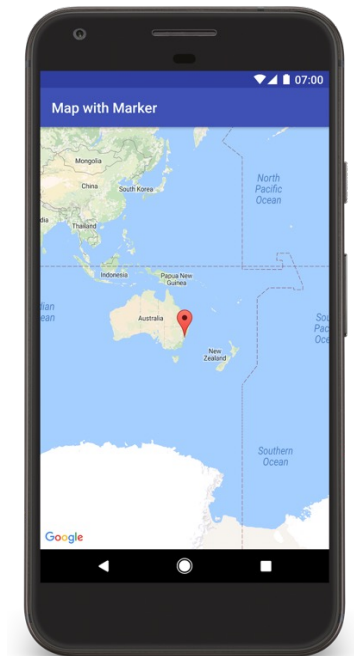
```
private suspend fun getFunction(key : Long): MyEntity?
{
    var myDBelement = databaseDao.get(key)
    return myDBelement
}
```

3. launch coroutine with the appropriate scope

```
private fun longDbWork(key: Long) {
    viewModelScope.launch {
        myDBelement = getFunction(key)
    }
}
```

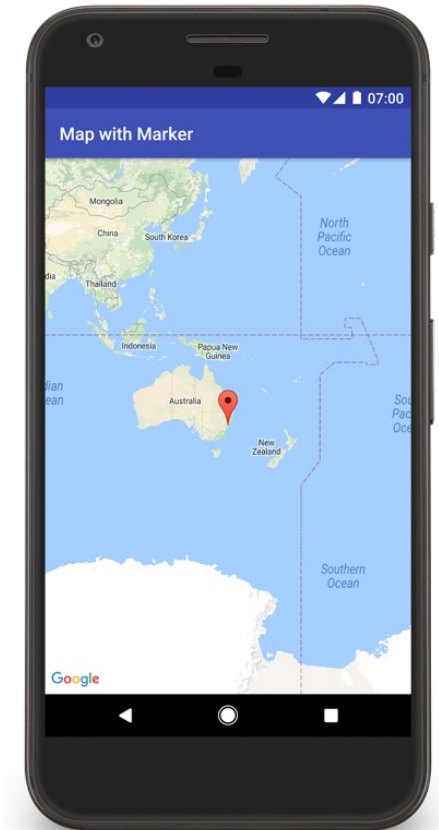
Outline of the class

- Coroutines and Room
- **Geolocation**
 - GoogleMaps API
 - Location system service



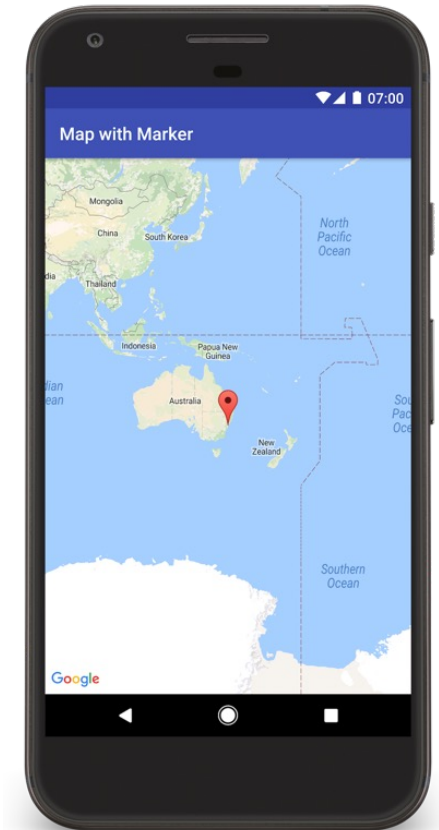
The GoogleMaps API

- The API allows you to add maps to your app based on Google Maps data.
- Takes care of:
 - Access Google maps servers
 - Data downloading
 - Map display
 - Touch gestures on the map.
- De-facto monopoly
 - Alternatives: <https://wiki.openstreetmap.org/wiki/Android>



Setting up GoogleMaps in Cloud Store

- GoogleMaps requires an API key
 - obtained from Google Cloud Console: console.cloud.google.com/
 - requires a billing method, even if GoogleMap API is free for use in GoogleMap composables
<https://developers.google.com/maps/billing-and-pricing/pricing#maps-sdk>



Displaying a map

1. Add a Google map key to the app manifest XML (obtained from GoogleCloud)

```
<meta-data
  android:name="com.google.android.geo.API_KEY"
  android:value="YOUR_KEY_HERE" />
```

2. Add a GoogleMap to the layout of the composable in which you want to host the map

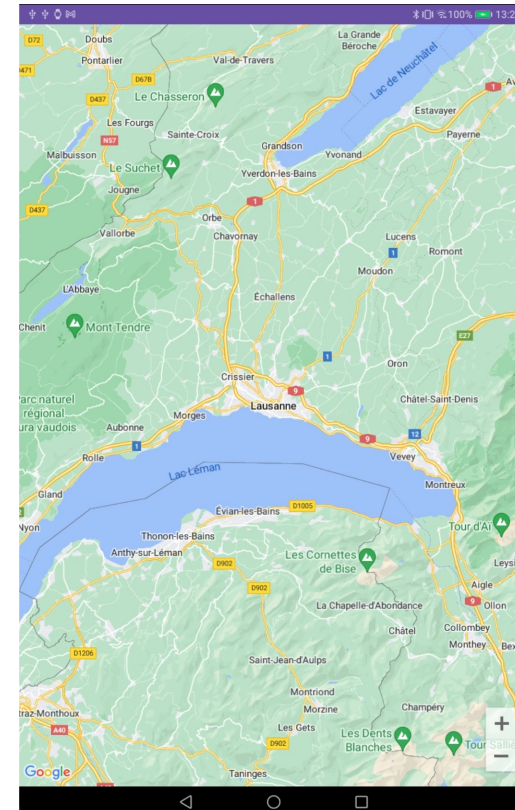
- Adding initial camera position

```
GoogleMap(
```

```
  modifier = Modifier.fillMaxSize(),
  cameraPositionState = cameraPositionState
)
```

```
val lausanne = LatLng(46.5197, 6.6323)
```

```
val cameraPositionState = rememberCameraPositionState {
  position = CameraPosition.fromLatLngZoom(lausanne, 10f)
}
```



Customizing the map: Zoom

Zoom levels

- 1 → World
- 5 → Continent
- 10 → City
- 15 → Streets
- 20 → Buildings

`position = CameraPosition.fromLatLngZoom(lausanne, 10f)`



Zoom level
5



Zoom level
15



Zoom level
20

Customizing the map: Type

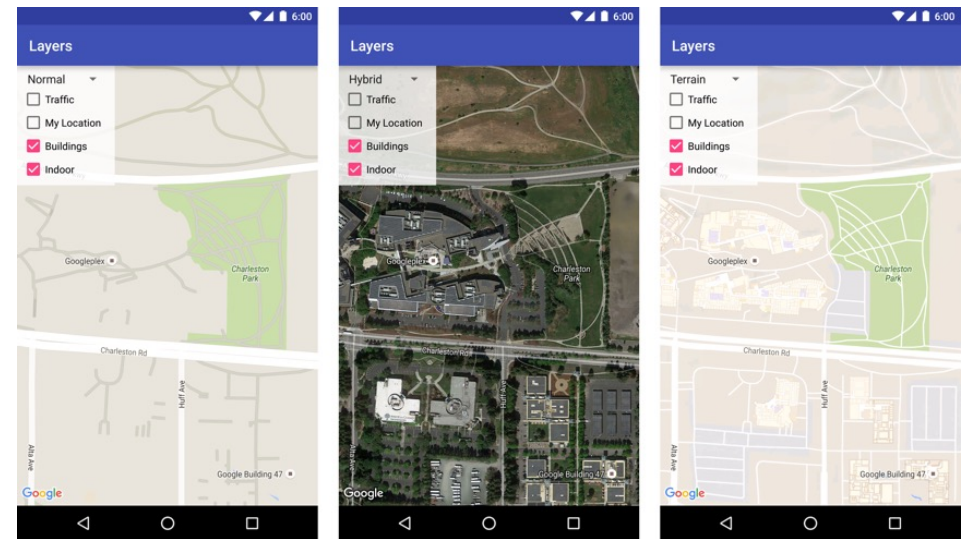
- Define the Map type, governing the overall representation of the map

- Normal** → Typical road map
- Hybrid** → Satellite data + roads
- Satellite** → Satellite data only
- Terrain** → Topographic data
- None** → no tiles, empty grid

GoogleMap(

...

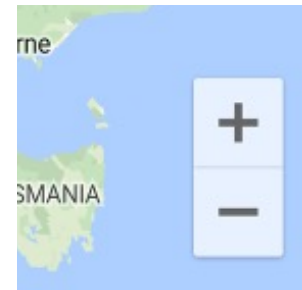
```
properties = MapProperties(mapType = MapType.NORMAL)
)
```



Customizing the map: Map controls

- Add zoom buttons

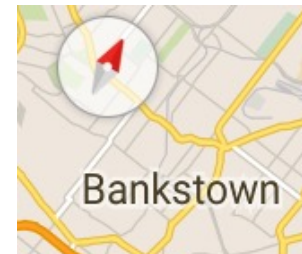
```
GoogleMap(  
  ...  
  uiSettings = MapUiSettings(zoomControlsEnabled = true)  
)
```



- Add compass

- appears when you rotate the map, or the map is not aligned to the North

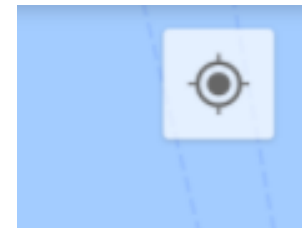
```
uiSettings = MapUiSettings(compassEnabled = true)
```



- Add myLocation button

- requires location information!

```
uiSettings = MapUiSettings(myLocationButtonEnabled = true)
```





Location data

- Apps must advertise the use of location data

```
<manifest xmlns:android= ... >
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
  <application>          ...
</application>
</manifest>
```

- ...ask the user permission to use it...

```
val cameraPermissionState = rememberPermissionState(
    android.Manifest.permission.ACCESS_FINE_LOCATION
)
if (!cameraPermissionState.hasPermission) {
    cameraPermissionState.launchPermissionRequest()
} else {
    //update UI accordingly
}
```

- ...and provide gradle dependencies...

```
implementation("com.google.accompanist:accompanist-permissions:0.23.1")
```

Location Provider(s)

- Location data can be obtained via several sources:
 - GPS, WiFi, Cell tower...



- Google provides a **FusedLocationProvider** system service
 - Provides best position estimate, without having to explicitly manage different sources

```
fusedLocationProviderClient = LocationServices
    .getFusedLocationProviderClient(context)
```



Retrieve location data

- Location data is retrieved by asking the location provider for the last known location

```
private fun getDeviceLocation() {
    try {
        fusedLocationProviderClient.lastLocation
            .addOnCompleteListener(this) { task ->
                if (task.isSuccessful) {
                    lastKnownLocation = task.result
                    if (lastKnownLocation != null) {
                        ... //Do something with the location
                    }
                }
            }
    } catch (e: SecurityException) {
        Log.e("Exception: %s", e.message, e)
    }
}
```

"Location provider, get me the last known location"

Listening for the Location provider replay

"Here it is!"

No permission to use location data



Request location updates

- Get location information at regular intervals

1. Create a Location Request

```
val locationRequest = LocationRequest.create()  
locationRequest.interval = 10000  
locationRequest.fastestInterval = 5000  
locationRequest.priority = LocationRequest.PRIORITY_HIGH_ACCURACY
```

2. Request location updates to the fusedLocationProvider

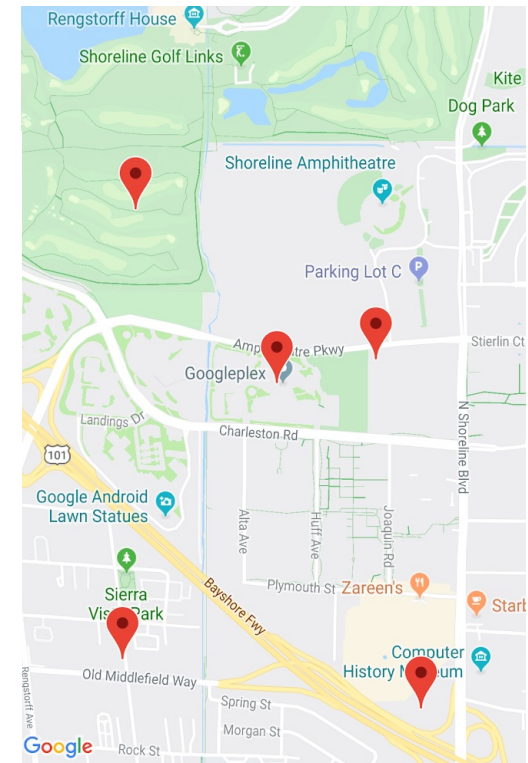
```
fusedLocationProviderClient  
    .requestLocationUpdates(  
        locationRequest, locationCallback, Looper.getMainLooper())
```



Request location updates

3. Implement the callback
→ what to do when data is received

```
private lateinit var locationCallback: LocationCallback
override fun onCreate(...) {
    ...
    locationCallback = object : LocationCallback() {
        override fun onLocationResult(locationResult: LocationResult?) {
            locationResult ?: return
            for (location in locationResult.locations){
                // Update UI with location data
                // ...
            }
        }
    }
}
```

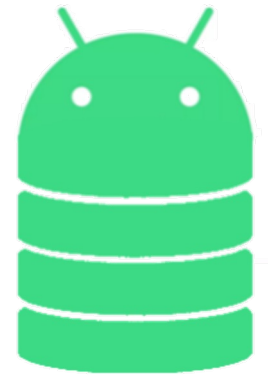
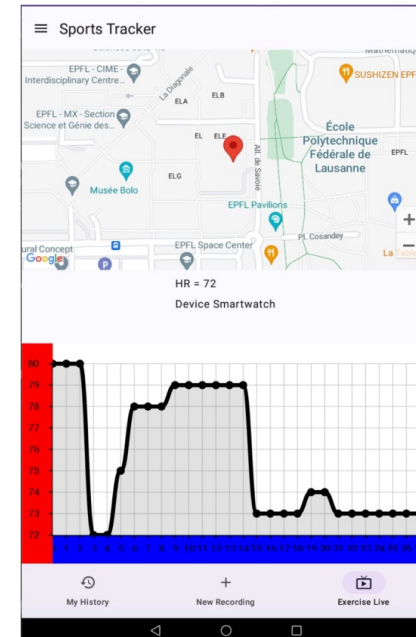




Today's Lab

- Add a map to ExerciseLiveScreen
 - Showing the user's current location

- Implement a Room database on the watch
 - storing and retrieving Heart Rate data



Questions?

