Projects Criteria

1) It needs to design some UI (at least three activities)

2) Needs to have some interaction with sensors

3) Needs to have some handling of data ~~on~~ using databases/data connectivity or over the network.

How background work is done by communicating across apps ⇒ Intent, Service and Notification.

Threads ⇒ Java has good support for threads.

UI ⇒ Main Thread/UI Thread.

⇒ slower in practice than anything related to UI

⇒ No accesses to file or database or network through this thread.

⇒ Android expects responsive UIs.

Threads was often used

{ ⇒ Hard to use developers. Need manual track of the state of each thread.

⇒ Thread 1. resume () {
    if status (Thread 2) == RUNNING:
}

Java ⇒ Threads are part of the language.
(helps avoid race conditions)

How?

```
synchronized {
    :
    :
    :
}
```

No variable can be accessed in any other thread if it is accessed within the synchronized block.

⇒ Creating and destroying threads have an overhead. (because requires going to kernel mode).

What is the alternative to threads?

⇒ Coroutines ⇒

Routines ⇒ Functions

Subroutines ⇒ Subfunctions { (Enter the execution of the function via one point, and run it until it returns)

⇓ Traditional way of programming ; may or may not be suitable.

Coroutines ⇒ (running of the called and calling function in handshake mode)

The called function should be able to suspend execution and then wakeup ; on getting some signal.

Android's coroutines :⇒ A coroutine builder to create a new coroutine.

Many of the classes used have a CoroutineScope class already defined

⇓

define its launch function to create a coroutine.

```
class ViewModelScope {
    ViewModelScope.launch { . . . }
}
```

→ coroutine

```
class Activity {
    lifeCycleScope.launch {. . . . }
}
```

How to make the subfunctions aware of the fact that it can be suspended?

If a function can resume at the same point without giving a wrong result, then it can be utilized as a co-routine.

(Should not use global variables; should not access resources like files where information has changed)

```
suspend fun abc (          )
        Delay (500)
        Logger ("  . . . . ")
```

There is a Job class returned by the launch function.

When you have the onstop function called (on killing of an activity), you can call the Jobo cancel() function to clean up the suspended thread).

```
class ViewModelScope {
    ListView lv = new ListView
    Job job? = ViewModelScope.launch {
        /* read the data from file and
           populate lv */
    }
}
```

① Data is static; no updates are possible

② Clean-up or closing of files will not happen if the app is closed when this ViewModelScope's launch thread is run

```
class Activity {
    fun onStop() {
        job?.cancel()
    }
```

[how to remove this inconvenience]

```
if (var) != null
    then operate on <var>
```

Nullable variables

within the launch block itself,
register a function called

```
viewLifecycleOwner.repeatOnLifecycle(
        LifeCycle.State.STARTED) {
    /* clean up */
```

}

The above code within this registered block would be automatically run by onStop. This is to ensure that the code is more logically organized

---

How to utilize databases

⇒ Run SQL queries to actually use databases.

ER model (Entity Relationship)

Mapping of OO model with the ER model.

⇓

Pre-defined annotations of classes, such as @Entity and @PrimaryKey,

```
@Entity
data class C {
    @PrimaryKey val id: UID
    val Title: String

    ...

}
```

No need of additional mapping of class variables with the schema columns.

Access data from a DB ⇒ (Trad) Run queries.

⇒ This should be done within an interface annotated by @Dao (Data Access Object)

```
@Dao
interface CDao {
    @Query("Select * from CDB")
    suspend fun getC(): List<C>
```

```
@Database(entities = [C], version = 1)
abstract class CDB {...
}
```

Must be executed only once in an application
⇒ Singletons
⇓
Specific classes that must have a single existence within the application

```
class CApplication : Application() {
    /* Create a singleton */
```

Needs to be registered in the manifest file for it to be usable.

How do we show the content of the database on the listView??

Can I retrieve data from before?

⇒ <u>Register</u> a "Flow" class at the time of defining the Launch block. Flow class has the methods to directly get the data initially and update it whenever required

```
suspend fun getResult : Flow<List<C>>
                 = CDB.getC()
```

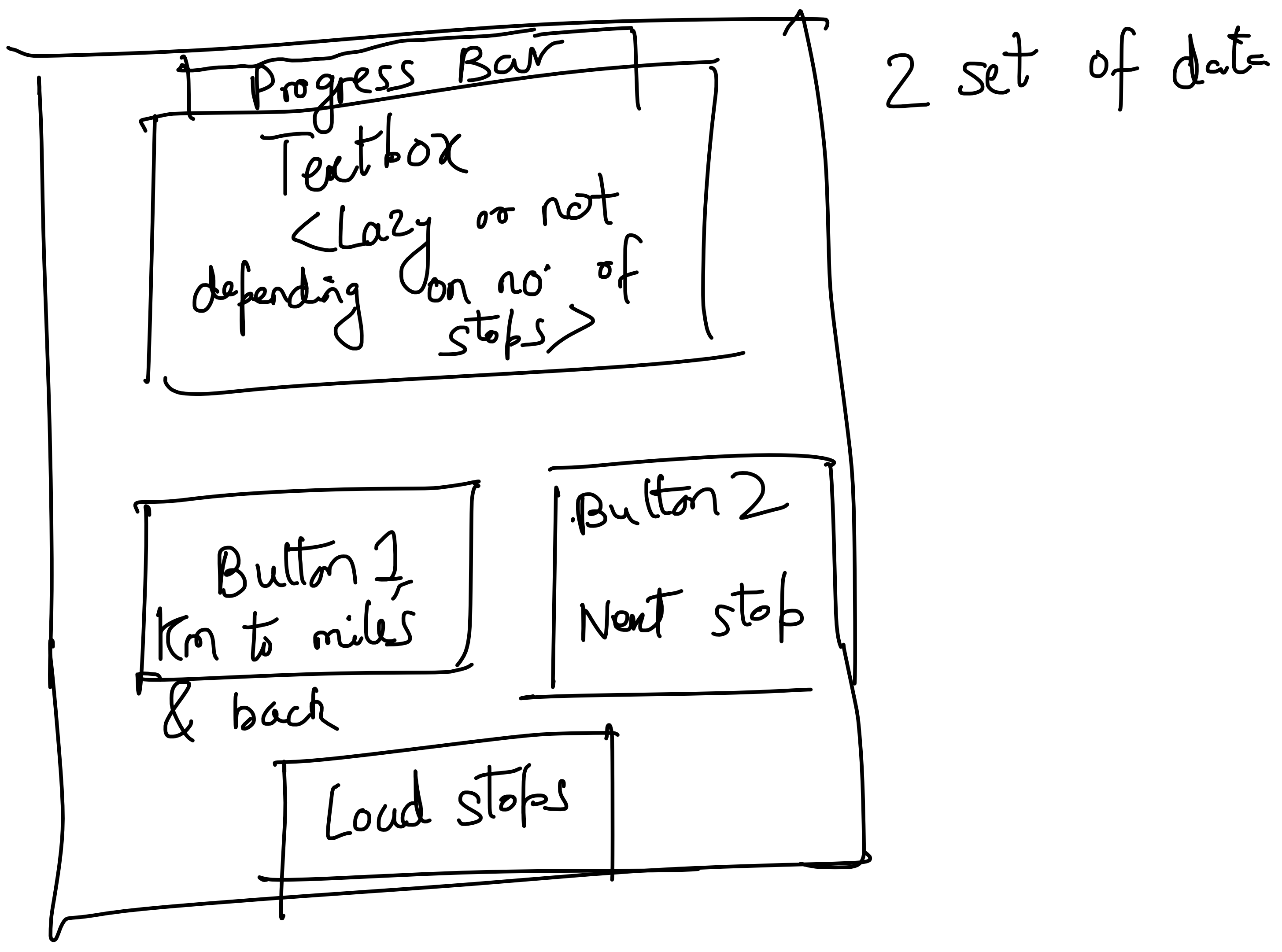
How are we using registering??

⇒ 1) • For cleanup, at the launch block

2) Keep updating the results from the DB.

SQLite Database by default

Repository pattern ⟹ Create classes for each entity and then additional classes to store the data as repository.


Progress Bar
Textbox
<Lazy or not depending on no. of stops>
Button 1, Km to miles & back
Button 2
Next stop
Load stops
2 set of data