



The
University
Of
Sheffield.



COM4510/6510

Software Development for Mobile Devices

Lab 5: Using Rooms

Temitope (Temi) Adeosun
The University of Sheffield
t.adeosun@sheffield.ac.uk



2 exercises

1. Takes your last Week's lab code, or provided [solution](#)
 - uploading thumbnails to the grid
 - Modify it so to use an asynchronous process so that each image is uploaded to the grid when available
2. Create a database using Room
 - To persist description, title and file path for every image
 - allow for insertion and deletion



The
University
Of
Sheffield.

But first...

out of memory error

Out of memory?

- Some of you will have noticed that if you load several large images, the app will crash
- This is because loading large images into memory to present only a reduced version of the image is wrong
 - it will take a lot of memory and Android will kill the app
- We should instead generate and display a thumbnail



The
University
Of
Sheffield.

Exercise 1a

How to generate a thumbnail

- Download the example solution code from last week
- Change MyAdapter as shown

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {  
    //Use the provided View Holder on the onCreateViewHolder method to populate the  
    // current row on the RecyclerView  
    if (items[position].image != -1) {  
        holder.imageView.setImageResource(items[position].image)  
    } else if (items[position].file != null) {  
        // val myBitmap = BitmapFactory.decodeFile(items[position].file?.file?.absolutePath)  
        // holder.imageView.setImageBitmap(myBitmap)  
  
        items[position].file?.file?.absolutePath?.let {  
            holder.imageView.setImageBitmap(MyAdapter.decodeSampledBitmapFromResource(it, 150, 150))  
        }  
    }  
  
    holder.itemView.setOnClickListener(View.OnClickListener {  
        val intent = Intent(context, ShowImageActivity::class.java)  
        intent.putExtra("position", position)  
        context.startActivity(intent)  
    })  
}
```

Before thumbnail

After thumbnail

And add...

- Add two helper functions the companion object declaration to resize the images

```
fun decodeSampledBitmapFromResource(filePath: String, reqWidth: Int, reqHeight: Int):
```

```
Bitmap {
```

```
    // First decode with inJustDecodeBounds=true to check dimensions
```

```
    val options = BitmapFactory.Options()
```

```
    options.inJustDecodeBounds = true
```

```
    BitmapFactory.decodeFile(filePath, options);
```

```
    // Calculate inSampleSize
```

```
    options.inSampleSize = calculateInSampleSize(options, reqWidth, reqHeight);
```

```
    // Decode bitmap with inSampleSize set
```

```
    options.inJustDecodeBounds = false
```

```
    return BitmapFactory.decodeFile(filePath, options);
```

```
}
```

```
private fun calculateInSampleSize(options: BitmapFactory.Options, reqWidth: Int, reqHeight: Int): Int {
```

```
    // Raw height and width of image
```

```
    val height = options.outHeight; val width = options.outWidth
```

```
    var inSampleSize = 1
```

```
    if (height > reqHeight || width > reqWidth) {
```

```
        val halfHeight = (height / 2).toInt()
```

```
        val halfWidth = (width / 2).toInt()
```

```
        // Calculate the largest inSampleSize value that is a power of 2 and keeps both
```

```
        // height and width larger than the requested height and width.
```

```
        while ((halfHeight / inSampleSize) >= reqHeight
```

```
            && (halfWidth / inSampleSize) >= reqWidth) {
```

```
            inSampleSize *= 2;
```

```
        }
```

```
    }
```

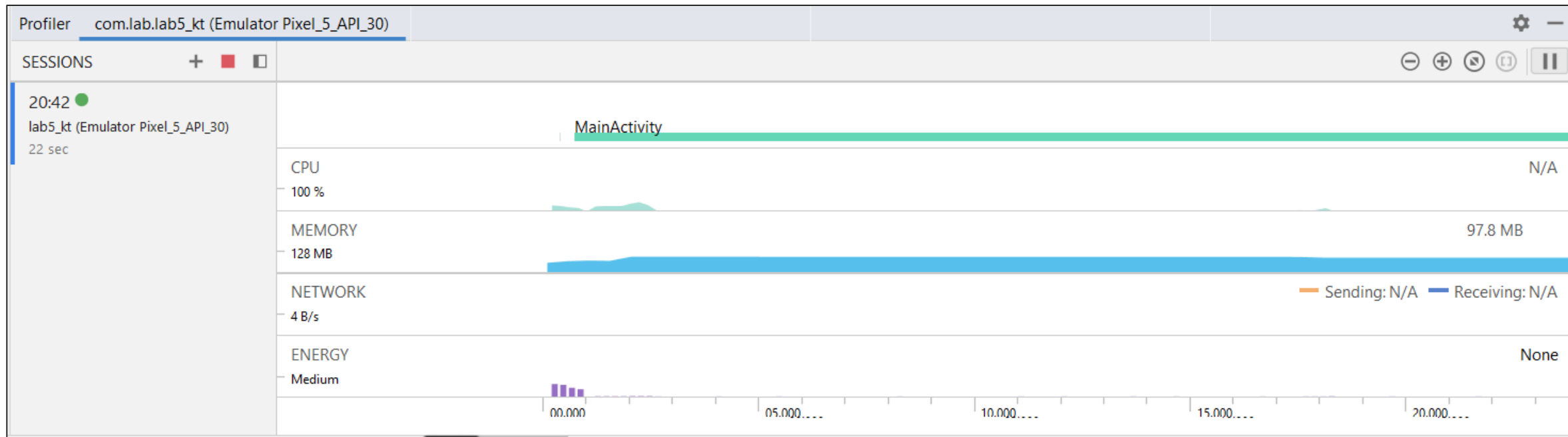
```
    return inSampleSize.toInt();
```

```
}
```

Effect

- Only the Thumbnail will be loaded into memory
- The app will not crash because:
 - Reduced memory usage. To check
 - Open profiler: View > Tool Windows > Profiler
 - Run using “Before thumbnail” and “After thumbnail” code. Add multiple images from the camera or the gallery by clicking the fab button
 - See the effect of “Before thumbnail” and “After thumbnail” on memory usage in profiler

Effect



App Profiler

- The [Lab 5 exercise 1b starter code](#) provided on Blackboard for today contains the changes in previous pages.
- Try it and make sure it works before using it for the next exercise!



The
University
Of
Sheffield.

Exercise 1b

Solving the bottleneck

Solving the bottleneck

- Given your program: think:
 - where is/are the bottleneck(s)?
 - where is/are the heaviest process(es)?
 - where is/are you blocking the UI Thread?
- how do you solve the bottleneck?
 - where do you need to insert an async process?
- Solution in the following slides **but try to answer the question yourself before flipping over**

The bottleneck

- The bottleneck is the loading of the file from the gallery and the creation of the bitmap
- In MyAdapter:

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {  
    //Use the provided View Holder on the onCreateViewHolder method to populate the  
    // current row on the RecyclerView  
    if (items[position].image != -1) {  
        holder.imageView.setImageResource(items[position].image)  
    } else if (items[position].file != null) {  
        // val myBitmap = BitmapFactory.decodeFile(items[position].file?.file?.absolutePath)  
        // holder.imageView.setImageBitmap(myBitmap)  
  
        items[position].file?.file?.absolutePath?.let {  
            holder.imageView.setImageBitmap(MyAdapter.decodeSampledBitmapFromResource(it, 150, 150))  
        }  
    }  
}  
...  
}
```

Solving the Bottleneck

- Now that you have identified the bottleneck
 - How do you solve it?
 - Normally, we'd use [AsyncTask](#)
 - An async task will require us to create a class that implements the AsyncTask interface, which requires 3 functions:
 - doInBackground
 - onProgressUpdate
 - onPostExecute
 - AsyncTask is now deprecated. Had some issues
 - Thread creation is resource intensive
 - Can lead to race conditions
 - Lots of effort to manage

Solving the Bottleneck

- Since we are using Kotlin, we can use Kotlin Coroutines for async. <https://www.youtube.com/watch?v=ne6CD1ZhAI0>
- Coroutines is not a new concept and are generalized subroutines for non-preemptive multitasking.
- Kotlin Coroutines:
 - supports asynchronous processing, concurrency, light weight multi threading and more
 - Simplifies async programming
 - Main safe async processing
 - reduces the need for implementing callback patterns (unlike AsyncTask or Executors)
 - conveniently converts async callback code into sequential code

Code changes

- Add Coroutine support to the app by adding dependency to the app gradle file:

```
implementation "org.jetbrains.kotlin:kotlinx-coroutines-core:1.5.2"
```

```
implementation "org.jetbrains.kotlin:kotlinx-coroutines-android:1.5.2"
```

- Update MyAdapter:

- Make decodeSampledBitmapFromResource() a suspending function

```
suspend fun decodeSampledBitmapFromResource ... {
```

- In onBindViewHolder, wrap the call to decodeSampledBitmapFromResource in a GlobalScope.launch

```
GlobalScope.launch(Dispatchers.Main){  
    val myBitmap = MyAdapter.decodeSampledBitmapFromResource(it,150, 150)  
    holder.imageView.setImageBitmap(myBitmap)  
}
```

Update: Changed to *Dispatchers.Main* from *Dispatchers.IO* because holder is updating a UI thread.

- See links sign-posted above for more about Coroutine and using Coroutine in Android
- Suspending functions must be called from other suspending functions
- `GlobalScope.launch` is suspending, but main safe - the calling function doesn't have to be suspending and it won't block the main thread.
 - It is called with the `Dispatchers.IO` option, which tells Coroutine to use the thread pool optimized for IO. Others include `Dispatchers.Default`, `Dispatchers.Unconfined`, `Dispatchers.Main`. See more about [dispatchers](#).
- Run the program
 - You may not notice a big difference without loading many images

Run the program

- You will see the effect of the async task:
 - images may appear in random order in the interface when you scroll
 - the interface should not lock (or nearly so)
 - see next slide



The
University
Of
Sheffield.

The other bottleneck

- If you load several images and try to load, you will see the interface is sluggish
 - images will appear slowly
 - sometimes scrolling is stopped
- Why is it the case?
 - see next slide but try to think before...

Building the thumbnail

- Building a thumbnail is a lot of work.
- However every time the bitmap is removed from the holder (because it is no longer visible) the bitmap is destroyed
- Try instead to keep the bitmap by writing it into a file
 - in this case you only create the thumbnail once
 - you will implement this as part of your Room implementation.

It is important that you do so when loading images in your assignment!!

Section updated

Introduction to Room

It is recommended that you at least have a glance through the *Introduction to Room* section (slides 22 to 33 – all slides with green borders).

However, this section is optional. If you wish to, you may continue from slide 34, and optionally review this section later. (The slides are in java, but the same concepts apply, links to the Kotlin content is provided and we will still discuss Room in class)

However, you can refer back to this section for more information about Exercise 2's implementation.

What is Room

- Room is an Object Relational Mapping (ORM) library that is provided as a wrapper around SQLite.
- Maps relational database tables to objects used in your code.
- Is a part of Android Architecture components in the Android Jetpack. We will talk more about these next week.

Room Entities

Defining data using Room entities

When using the [Room persistence library](#), you define sets of related fields as *entities*. For each entity, a table is created within the associated [Database](#) object to hold the items.

By default, Room creates a column for each field that's defined in the entity. If an entity has fields that you don't want to persist, you can annotate them using [@Ignore](#). You must reference the entity class through the [entities](#) array in the [Database](#) class.

The following code snippet shows how to define an entity:

```
@Entity
class User {
    @PrimaryKey
    public int id;

    public String firstName;
    public String lastName;

    @Ignore
    Bitmap picture;
}
```

To persist a field, Room must have access to it. You can make a field public, or you can provide a getter and setter for it. If you use getter and setter methods, keep in mind that they're based on JavaBeans conventions in Room.

[< Previous](#)[Next >](#)

This lesson teaches you to

- > [Use a primary key](#)
- > [Annotate indices and uniqueness](#)
- > [Define relationships between objects](#)
- > [Create nested objects](#)

Primary Keys

Use a primary key

Each entity must define at least 1 field as a primary key. Even when there is only 1 field, you still need to annotate the field with the `@PrimaryKey` annotation. Also, if you want Room to assign automatic IDs to entities, you can set the `@PrimaryKey`'s `autoGenerate` property. If the entity has a composite primary key, you can use the `primaryKey` property of the `@Entity` annotation, as shown in the following code snippet:

```
@Entity(primaryKey = {"firstName", "lastName"})
class User {
    public String firstName;
    public String lastName;

    @Ignore
    Bitmap picture;
}
```

By default, Room uses the class name as the database table name. If you want the table to have a different name, set the `tableName` property of the `@Entity` annotation, as shown in the following code snippet:

```
@Entity(tableName = "users")
class User {
    ...
}
```

Caution: Table names in SQLite are case-insensitive.

Filed/Column Name

Similar to the `tableName` property, Room uses the field names as the column names in the database. If you want a column to have a different name, add the `@ColumnInfo` annotation to a field, as shown in the following code snippet:

```
@Entity(tableName = "users")
class User {
    @PrimaryKey
    public int id;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;

    @Ignore
    Bitmap picture;
}
```

Annotate indices and uniqueness

Depending on how you access the data, you might want to index certain fields in the database to speed up your queries. To add indices to an entity, include the `indices` property within the `@Entity` annotation, listing the names of the columns that you want to include in the index or composite index. The following code snippet demonstrates this annotation process:

```
@Entity(indices = {@Index("name"),  
                  @Index(value = {"last_name", "address"})})  
class User {  
    @PrimaryKey  
    public int id;  
  
    public String firstName;  
    public String address;  
  
    @ColumnInfo(name = "last_name")  
    public String lastName;  
  
    @Ignore  
    Bitmap picture;  
}
```

Unique elements

Sometimes, certain fields or groups of fields in a database must be unique. You can enforce this uniqueness property by setting the `unique` property of an `@Index` annotation to `true`. The following code sample prevents a table from having two rows that contain the same set of values for the `firstName` and `lastName` columns:

```
@Entity(indices = {@Index(value = {"first_name", "last_name"},  
    unique = true)})  
class User {  
    @PrimaryKey  
    public int id;  
  
    @ColumnInfo(name = "first_name")  
    public String firstName;  
  
    @ColumnInfo(name = "last_name")  
    public String lastName;  
  
    @Ignore  
    Bitmap picture;  
}
```

Joining tables

- Room is backed by SQLite, so it does not allow to have relations between objects
- It requires the definition of foreign keys



Because SQLite is a relational database, you can specify relationships between objects. Even though most object-relational mapping libraries allow entity objects to reference each other, Room explicitly forbids this. To learn about the technical reasoning behind this decision, see [Understand why Room doesn't allow object references](#).

Even though you cannot use direct relationships, Room still allows you to define Foreign Key constraints between entities.

For example, if there's another entity called **Book**, you can define its relationship to the **User** entity using the [@ForeignKey](#) annotation, as shown in the following code snippet:


```
@Entity(foreignKeys = @ForeignKey(entity = User.class,
                                   parentColumns = "id",
                                   childColumns = "user_id"))

class Book {
    @PrimaryKey
    public int bookId;

    public String title;

    @ColumnInfo(name = "user_id")
    public int userId;
}
```

Foreign keys are very powerful, as they allow you to specify what occurs when the referenced entity is updated. For instance, you can tell SQLite to delete all books for a user if the corresponding instance of **User** is deleted by including `onDelete = CASCADE` in the [@ForeignKey](#) annotation.

Note: SQLite handles [@Insert\(onConflict = REPLACE\)](#) as a set of **REMOVE** and **REPLACE** operations instead of a single **UPDATE** operation. This method of replacing conflicting values could affect your foreign key constraints. For more details, see the [SQLite documentation](#)  for the **ON_CONFLICT** clause.

Nested Objects

Create nested objects

Sometimes, you'd like to express an entity or plain old Java object (POJO) as a cohesive whole in your database logic, even if the object contains several fields. In these situations, you can use the `@Embedded` annotation to represent an object that you'd like to decompose into its subfields within a table. You can then query the embedded fields just as you would for other individual columns.

For instance, our `User` class can include a field of type `Address`, which represents a composition of fields named `street`, `city`, `state`, and `postCode`. To store the composed columns separately in the table, include an `Address` field in the `User` class that is annotated with `@Embedded`, as shown in the following code snippet:



```
class Address {  
    public String street;  
    public String state;  
    public String city;  
  
    @ColumnInfo(name = "post_code")  
    public int postCode;  
}  
  
@Entity  
class User {  
    @PrimaryKey  
    public int id;  
  
    public String firstName;  
  
    @Embedded  
    public Address address;  
}
```

The table representing a **User** object then contains columns with the following names: **id**, **firstName**, **street**, **state**, **city**, and **post_code**.

Note: Embedded fields can also include other embedded fields.

If an entity has multiple embedded fields of the same type, you can keep each column unique by setting the **prefix** property. Room then adds the provided value to the beginning of each column name in the embedded object.



Accessing data using Room DAOs

To access your app's data using the [Room persistence library](#), you work with *data access objects*, or DAOs. This set of [Dao](#) objects forms the main component of Room, as each DAO includes methods that offer abstract access to your app's database.

By accessing a database using a DAO class instead of query builders or direct queries, you can separate different components of your database architecture. Furthermore, DAOs allow you to easily mock database access as you [test your app](#).

A DAO can be either an interface or an abstract class. If it's an abstract class, it can optionally have a constructor that takes a [RoomDatabase](#) as its only parameter. Room creates each DAO implementation at compile time.

Note: Room doesn't support database access on the main thread unless you've called [allowMainThreadQueries\(\)](#) on the builder because it might lock the UI for a long period of time. Asynchronous queries, queries that return instances of [LiveData](#) or [Flowable](#), are supported on the main thread.

< Previous

Next >

This lesson teaches you to

- > [Define methods for convenience](#)
 - > [Insert](#)
 - > [Update](#)
 - > [Delete](#)
- > [Query for information](#)
 - > [Simple queries](#)
 - > [Passing parameters into the query](#)
 - > [Returning subsets of columns](#)
 - > [Passing a collection of arguments](#)
 - > [Observable queries](#)
 - > [Reactive queries with RxJava](#)
 - > [Direct cursor access](#)
 - > [Querying multiple tables](#)

Migrating

- When you change the database relations, you must migrate the database or the app will crash
- Method:
 - Auto-migration:
 - change db version (e.g. from 1 to 2)
 - Define a migration strategy
 - Manual migration

[see webpage](#)

```
Room.databaseBuilder(getApplicationContext(), MyDb.class, "database-name")
    .addMigrations(MIGRATION_1_2, MIGRATION_2_3).build();

static final Migration MIGRATION_1_2 = new Migration(1, 2) {
    @Override
    public void migrate(SupportSQLiteDatabase database) {
        database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, "
            + "`name` TEXT, PRIMARY KEY(`id`))");
    }
};

static final Migration MIGRATION_2_3 = new Migration(2, 3) {
    @Override
    public void migrate(SupportSQLiteDatabase database) {
        database.execSQL("ALTER TABLE Book "
            + " ADD COLUMN pub year INTEGER");
    }
};
```

Exercise 2a

Implementing Room

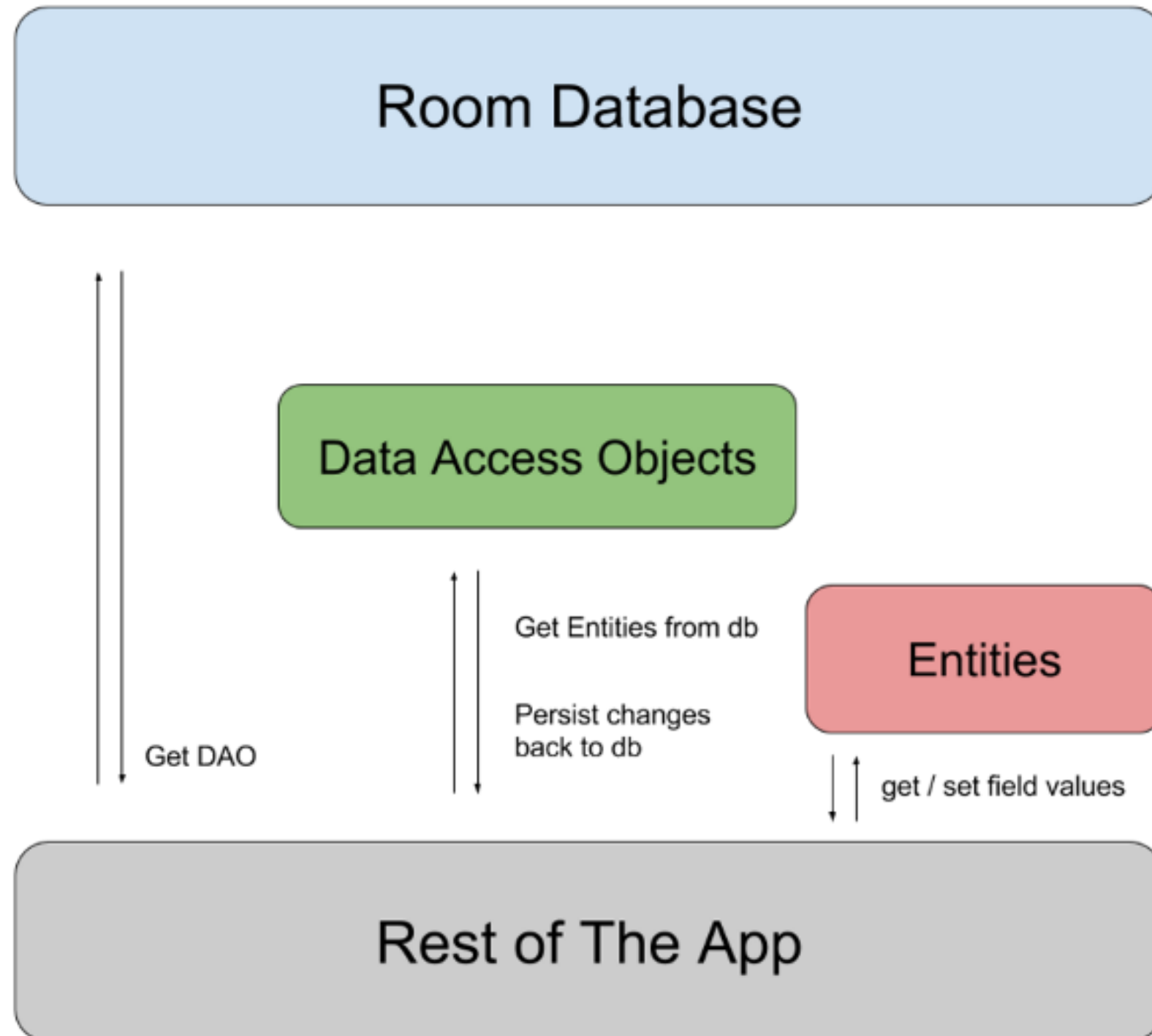
- In this exercise you will add support for Room to the application.

Room components

- Room is an Object Relationship Mapping Library
- Has three main components:
 - Data Entities: a data entity is a class that represents a table in a database. The fields map to table columns.
 - Data Access Objects (DAO): class that provides database access operation (select, update, insert, delete) of data entities from the database tables
 - Database class: provides an abstraction of the database and provides connection to the underlying app database (SQLite)

Room components

- Component interaction with Mobile app



Implementation

- Continue with the code in the state it was at the end of Exercise 1b
- Intended database design
- Enable Room:
 - Add Room libraries to the gradle app file (version 2.3.0 is current):


```
implementation "androidx.room:room-runtime:2.3.0"  
kapt "androidx.room:room-compiler:2.3.0"  
implementation "androidx.room:room-ktx:2.3.0"
```
 - To use the kapt library, add the Kotlin kapt plugin in the app gradle build file (plugin section):

```
id 'kotlin-kapt'
```

Kapt is the Kotlin annotation processing tool. Needed for the Room annotations

Implementation

- Intended database design

Live updates 

id	title	uri	thumbnailUri	description
16	ei_1635455456602.jpg	/data/user/0/com	NULL	Loren ispum
18	ei_1635456538591.jpg	/data/user/0/com	NULL	Loren ispum
27	ei_1635459171574.jpg	/data/user/0/com	NULL	NULL
28	ei_1635459180557.jpg	/data/user/0/com	NULL	NULL

Create the Entity

- Create a package called ***data*** under the ***com.lab.lab5_kt*** package:
 - right click com.lab.lab5_kt: New > Package
- To the data package, add a kotlin data class, name it ***ImageData***
 - *Declare parameters for ImageData's default constructor with name and type that agree with the intended database table as shown on previous page:*

```
data class ImageData(  
    var id: Int = 0,  
    val imageUri: String,  
    var imageDescription: String? = null,  
    val thumbnailUri: String  
)
```

The “data” keyword makes this a data class

Create the Entity (cont.)

- To make **ImageData** an entity, apply the @Entity annotation

```
@Entity(tableName = "image")
```

```
data class ImageData(  
...
```

@Entity can take a couple of arguments, of which *tableName*, *indices* respectively specified the name of the table in the database and the fields use for database indexing

- Apply annotations to the fields
 - @PrimaryKey(autogenerate = true) to the id field.
Marks field as primary key and ensures the id field is auto generated when a new ImageData entity is inserted into the image table.
 - @ColumnInfo(name = "preferred_name") to the other fields.
This allows us to provide a preferred name for the table columns. Otherwise, the field name will be used.
@ColumnInfo is optional, but a good practice:
 - allows us to change the field name later, without changing the column name.
 - database schemas use a different naming convention from application code, so this separation is good.

Create the Entity (cont.)

- @Ignore for the thumbnailImagePath and image fields, tells Room to ignore creating a column for these field.

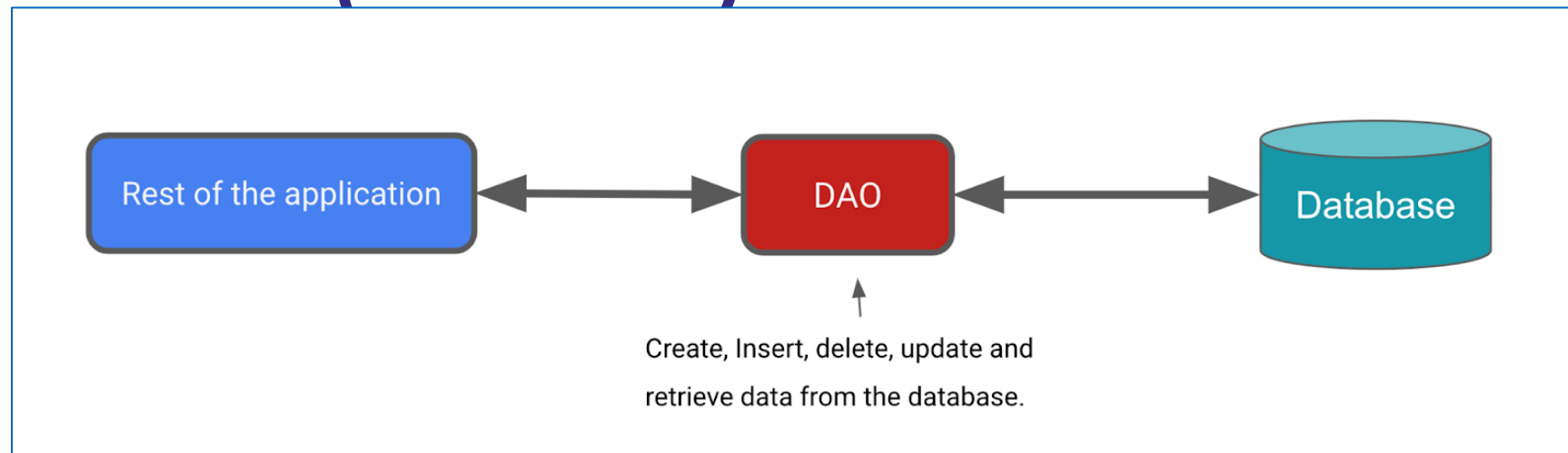
- The final ImageData entity should look like this:

```
@Entity(tableName = "image", indices = [Index(value = ["id", "title"])])  
data class ImageData(  
    @PrimaryKey(autoGenerate = true) var id: Int = 0,  
    @ColumnInfo(name="uri") val imageUri: String,  
    @ColumnInfo(name="title") var imageTitle: String,  
    @ColumnInfo(name="description") var imageDescription: String? = null,  
    @ColumnInfo(name="thumbnailUri") var thumbnailUri: String? = null,  
{  
    @Ignore  
    var thumbnail: Bitmap? = null  
}
```

Note: imageDescription is nullable because description are optional.

Note: imageTitle and originalPath are specified in the default constructor

Data Access Object (DAO)



- Implemented as an Interface – Room creates the class that implements the Interface to perform the actual data access. Room's implementation is based on the annotation configuration and specified types.
- Provides functions for database operations – retrieval, update, insert, delete, which are respectively annotated `@Query`, `@Update`, `@Insert`, `@Delete` and `@RawQuery`. `@Query` can also be used for other types of queries (i.e. joins)

Creating the DAO

- Operations needed – for this app, we need to:
 - *Get all ImageData entities*, i.e. when the app first loads to display all images that were previously added.
 - *Get a specific ImageData entity given it's id*
 - *Insert ImageData entities* as they are added to application
 - *Update image description*
 - *Delete a given ImageData entity from the database*

Creating the DAO

- In the data package, create a kotlin Interface named ImageDataDao, annotated it @Dao as shown:

```
@Dao  
interface ImageDataDao {  
  
}
```

- Add function declarations for each of the required operations:
 - Use suspending functions for operations that are potentially long running operations
 - Room uses Coroutine in the background for the suspending functions.
 - See next page for DAO implementation
 - Note @Insert annotation's *onConflict* parameter – specified the strategy for responding when two entities have the same id. See [here](#) for more conflict strategy options.

Creating the DAO

- ImageDataDao

@Dao

```
interface ImageDataDao {
```

```
    @Query("SELECT * from image ORDER by id ASC")
```

```
    suspend fun getItems(): List<ImageData>
```

```
    @Query("SELECT * from image WHERE id = :id")
```

```
    fun getItem(id: Int): ImageData
```

```
    @Insert(onConflict = OnConflictStrategy.REPLACE)
```

```
    suspend fun insert(singleImageData: ImageData): Long
```

```
    @Update
```

```
    suspend fun update(imageData: ImageData)
```

```
    @Delete
```

```
    suspend fun delete(imageData: ImageData)
```

```
}
```

Create the Database

- Implemented as a Kotlin abstract class – Room provides the concrete class
- Marked with the `@Database` annotation
- Extends the *RoomDataBase* Interface.
- Create it in the *data* package, name it **ImageRoomDatabase**

```
@Database(entities = [ImageData::class], version = 1, exportSchema = false)
abstract class ImageRoomDatabase: RoomDatabase() {
}
```

- Note about `@Database` parameters:
 - entities – specified the class of the list of entities this database will handle, this only handles one entity
 - version – increase this to kickstart migration whenever the entity changes needs a new database schema
 - exportSchema – set to false to prevent keeping schema history (not needed here)

Create the Database (cont.)

- ImageRoomDatabase:
 - needs to know about the DAO it is working with:
 - Add an abstract function to return ImageDataDao

```
abstract fun imageDataDao(): ImageDataDao
```
 - returns an instance of a database –only one such instance should exist in the application irrespective of the number of activities where the database instance is “created”. So, we need to implement a singleton pattern.
 - In Kotlin, you need a companion object
 - Inside the companion object, declare private nullable variable, INSTANCE – initialize to null.
 - INSTANCE variable is the reference to the ImageRoomDatabase object if one has been instantiated. Created once only if INSTANCE is null.
 - Annotate INSTANCE with @Volatile, to prevent caching the data, and immediate read/write

Create the Database (cont.)

- ImageRoomDatabase:
 - in the companion object, we need a function that can return INSTANCE if it is not null, or instantiate and return it if it is null. Call this function ***getDatabase()***. The companion object now looks like this:

```
companion object{
    @Volatile
    private var INSTANCE: ImageRoomDatabase? = null

    fun getDatabase(context: Context): ImageRoomDatabase{
        return INSTANCE ?: synchronized(this){
            val instance = Room.databaseBuilder(
                context.applicationContext,
                ImageRoomDatabase::class.java,
                "lab5_database"
            )
                .fallbackToDestructiveMigration()
                .build()
            INSTANCE = instance
        }
        return instance
    }
}
```

Create the Database (cont.)

- ImageRoomDatabase:
 - Read and make sure you understand what is happening in the above code before implementing in ImageRoomDatabase. Ask questions if you don't understand.
 - Synchronized – creates a lock to prevent race condition where multiple threads can call `getDatabase` potentially resulting in more than once `INSTANCE` object.
 - `ImageRoomDatabase::class.java` – the type of `RoomDatabase` object that must be returned
 - `context.applicationContext` – the context which the `ImageRoomDatabase` will be associated with. We are using the application context, so the database doesn't get destroyed between Activity/Fragment lifecycle changes.
 - “lab5_database” – the name the database will be called in SQLite

Create the Database (cont.)

- ImageRoomDatabase:
 - Room.databaseBuider – returns a [RoomDatabase.Builder](#), which is able connect to a database if it exists or creates it if it does not. Note – this supports the [builder pattern](#).
 - fallbackToDestructiveMigration() – where migration is required, this strategy deletes the existing database and creates a new one (data loss will result, doesn't matter in this example). See [here](#) for more about Migration.
 - Builder.build() – where migration is required, this strategy deletes the existing database and creates a new one (data loss will result, doesn't matter in this example). See [here](#) for more about Migration.
 - Note: several functions can be called on the Builder. See the [Builder documentation](#) for other functions.

Instantiate RoomDatabase object

- The implementation of the Room for this application is nearly complete.
 - All that is needed now is to call `ImageRoomDatabase.getDatabase()`
- But where should we call `ImageRoomDatabase.getDatabase()?!?`
- Consider Activity/Fragment changing lifecycle – database should live in the Application context, not an Activity/Fragment
 - call `getDatabase` from within the Application, [see also](#).
- So, `getDatabase()` goes in an Application class

Implements a custom Application class

- In the `com.lab.lab5_kt` package, create a class `ImageApplication` which extends the `Application` class.

```
class ImageApplication: Application() {  
}
```

- This class inherits from `Application` class and can replace the default `Application` class invoked by the framework on start-up.
- To tell Android to run your app using this `Application` class:
 - Update `AndroidManifest` and add in the `<application>` tag:

```
android:name="com.lab.lab5_kt.ImageApplication"
```

Instantiate RoomDatabase object (cont.)

- In ImageApplication class declare ***databaseObj***
- Instantiate it by calling getDatabase with the ***lazy*** delegate
 - lazy allows ***databaseObj*** to be created on demand, the first time the variable is accessed, rather than when the app starts

```
class ImageApplication: Application() {  
    val databaseObj: ImageRoomDatabase by lazy { ImageRoomDatabase.getDatabase(this) }  
}
```

- **Run your app** to ensure everything is good – it should run OK, creating the Room classes in the build directory – try Ctrl + click on ImageData class name to see some autogenerated code
 - but no behaviors changes yet because ***databaseObj*** has not been used yet.
 - Confirm you do not have a database yet in the Database Inspector: In menu, use View > Tool Windows > App Inspection

Exercise 2b

Persist data with Room

- In this exercise you will persist the description, title, file path of every image
- Download Lab 5 exercise 2b starter code and images.zip file. This starter code contains the code changes from exercise 1a to 2a. It also includes changes you need to get started with exercise 2b.

What has changed?

Check all changes and make sure you understand.

Layouts & UI design:

- Added new Activity: `EditActivity.kt` and `activity_edit.xml` layout file.
- Renamed layout files:
 - `activity_message2.xml` > `activity_show_image.xml`
 - `Activity_camera.xml` > `activity_gallery.xml`
- Edited layout file `activity_show_image.xml`:
 - Changed layout from `CoordinatorLayout` to `ConstraintLayout`
 - Added Fab button *fab_edit*
- *Updated the color theme – just for variation*

What has changed?

- Removed ImageElement.kt class file
- Edits to MyAdapter:
 - Changed all List<ImageElement> reference to MutableList<ImageData>
 - Update onBindViewHolder() to use ImageData instead of ImageElement object (old code commented out so you can compare)
 - Declared and used a dedicated Coroutine scope variable, instead of GlobalScope – still the same effect since they both use Dispatchers.Main

What has changed?

- Edited MainActivity:
 - Changed myDataset type from `MutableList<ImageElement>` to `MutableList<ImageData>`, and make it a *lateinit var*
 - Renamed `getImageElement()` to `getImageData()` and updated it to use the `ImageData` data class, instead of `ImageElement` (original content commented out so you can compare)
 - `getImageData()` calls `insertData()` – the body of this `insertData()` is not yet implemented.
 - An `ImageDataDao` object has been declared
`private lateinit var daoObj: ImageDataDao`
 - Next, you will make the first call to the database
 - Declared `daoObj` as `ImageDataDao`
 - The content of `initData` has been removed, so the app no longer loads the sample images provided.
Next, we will load data from the database

Initiate the first call to Room?

- Before making changes, run the code sample you have been provided, open the App Inspector and confirm there is no database installed on the AVD.
- Edit `initData()`
 - Instantiate `daoObj` by assigning it to the instance of `ImageRoomDatabase` in `ImageApplication`
`daoObj = (this@MainActivity.application as ImageApplication).databaseObj.imageDataDao()`
 - Now you can make a call to retrieve data from the database. Below this assignment make a call to `daoObj.getItems()`
`var data = daoObj.getItems()`
 - Then update `myDataset` variable with this:
`myDataset.addAll(data)`

Initiate the first call to Room?

- Observe the error message – this is because `getItems()` was declared a suspending function – which can only be called from another suspending function or a Coroutine.
- To fix the error, wrap the call to the `getItems()` with a coroutine:

```
GlobalScope.launch {  
    daoObj = (this@MainActivity.application as ImageApplication)  
        .databaseObj.imageDataDao()  
    myDataset.addAll(daoObj.getItems())  
}
```

- Now, run the project again and check that a database has been created by checking in the App Inspector.
- You have Room setup and performed one database operation – retrieving data.

More

- The solution file contains the current implementation and additional code for insert, update, delete.
- Please go through the solution code and engage in the discussion board if you have questions.
- Note the following:
 - The solution uses MVC – we will look at MVVM later
 - The solution requires a lot of boiler plate code to make support the flow of data between Activities.