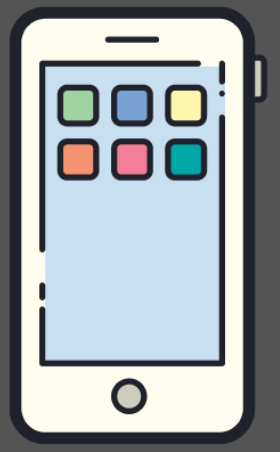


MOBILE DEVELOPMENT

Local DB (SQLite)

Overview

- Recall: Saving Data (SharedPreferences)
- SQLite (Local DB)
- Design Considerations
- Reminders

RECALL: Saving Data

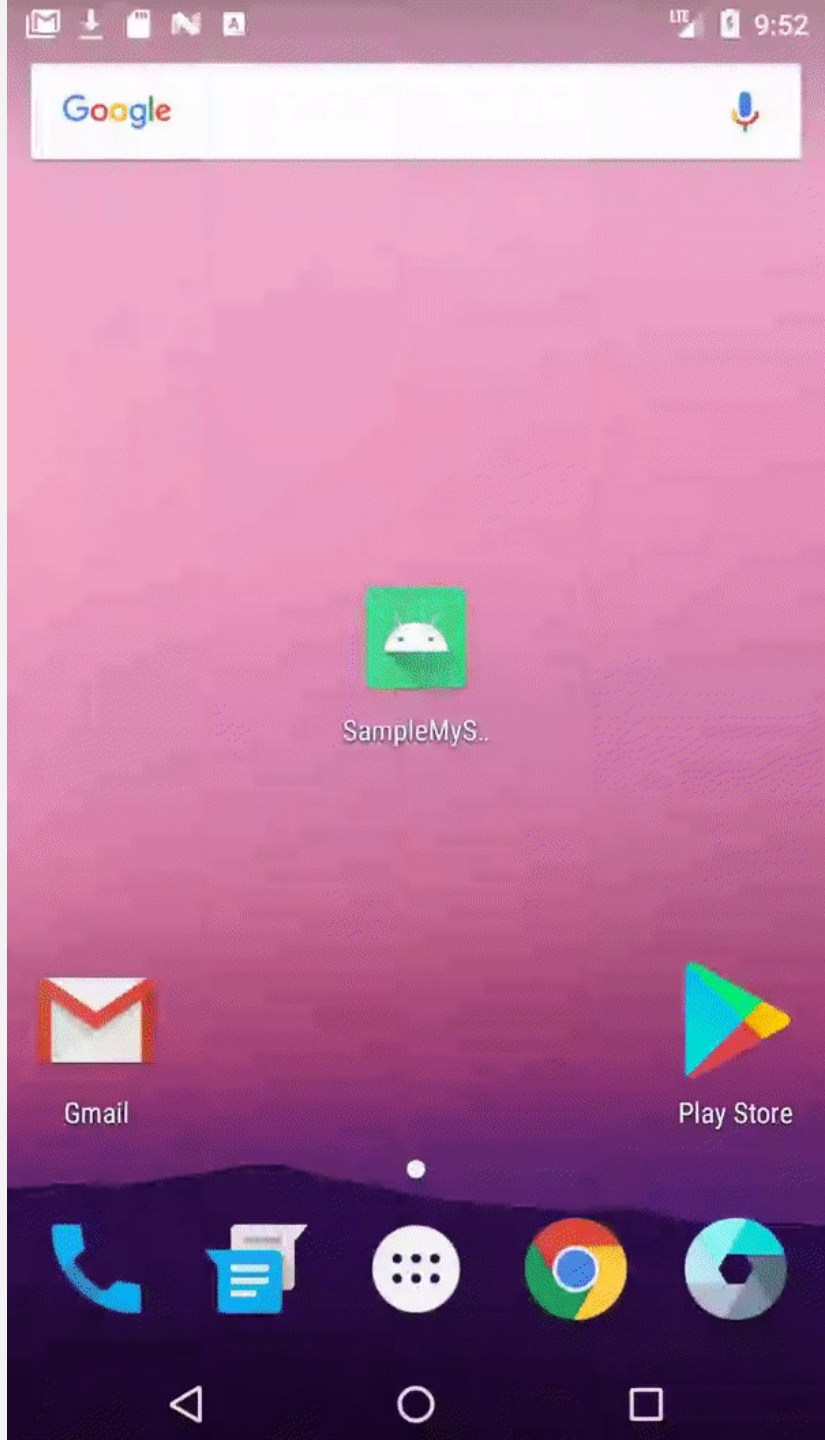
- There are many ways one can save data in Android
  - Database (local / online)
  - Files
  - SharedPreferences

- For now, we'll focus on SharedPreferences

Let’s say…

- We had to develop an app that could save Contacts, as seen to the side
- Our current task is to make sure the data persists even after closing the app

- What do we do?



We can try to apply SharedPreferences…

```
private void saveToSP(ArrayList<Contact> contacts) {
    // Assuming insert is always made at the last element in the ArrayList
    int newElement = contacts.size();

    SharedPreferences.Editor editor = PreferenceManager.getDefaultSharedPreferences(this).edit();
    editor.putString(FIRST_NAME_KEY + newElement, contacts.get(newElement).getFirstName());
    editor.putString(LAST_NAME_KEY + newElement, contacts.get(newElement).getLastName());
    editor.putString(NUMBER_KEY + newElement, contacts.get(newElement).getNumber());
    editor.putString(IMAGE_URI_KEY + newElement, contacts.get(newElement).getImageUri());
    editor.putInt(TOTAL_CONTACTS_KEY, newElement);
    editor.apply();
}
```

Saving / Writing

Reading

```
private ArrayList<Contact> readAllContacts() {
    ArrayList<Contact> contacts = new ArrayList<>();

    SharedPreferences sp = PreferenceManager.getDefaultSharedPreferences(this);
    int totalContacts = sp.getInt(TOTAL_CONTACTS_KEY, 0);
    for(int i = 1; i <= totalContacts; i++) {
        contacts.add(new Contact(
                sp.getString(FIRST_NAME_KEY + i, null), sp.getString(LAST_NAME_KEY + i, null),
                sp.getString(NUMBER_KEY + i, null), sp.getString(IMAGE_URI_KEY + i, null))
        );
    }

    return contacts;
}
```

Basically, utilize the current element's spot as an id for the Contact

However… **RECALL**: SharedPreferences

- Allow for data to persist on disk after the application is destroyed
  - Saved as an XML file (not a resource, but a file)
- Stores key-value pairs of primitive data types
  - Similar to Bundles
- Not good for storing complex data

SharedPreference is horrible for saving vertically scaling data

RECALL: Saving Data

- There are many ways one can save data in Android
  - Database (local / online)
  - Files
  - ~~SharedPreferences~~

- ~~For now, we'll focus on SharedPreferences~~

Database

- A database is an organized collection of data
  - Sometimes requires a strict structure (SQL)
  - Sometimes the structure can be flexible (NoSQL)

- We'll eventually discuss NoSQL next week, but for now we focus on SQL

Database – SQLite

- Android devices utilize SQLite for storing structured data
  - C-based
  - Is a RDBMS that utilizes SQL

```
SELECT * FROM Artists;

INSERT INTO Albums (AlbumName, ArtistId) VALUES ('Ziltoid the Omniscient',  '12');

DELETE FROM Artists;
```

Database – SQLite

- So we can have a little structure to how we'll learn SQLite, we'll implement the Contacts app we saw awhile ago
- Zip to a bare project can be found in module 4.a
  - Implementation only includes: UI, RecyclerView, Adding to RecyclerView

- Let's take a look at the base app first ☺

Implementation

- If you look up how to implement SQLite, you'll find that there are a lot of different ways to implement it
  - Most solutions differ in terms of efficient handling

- Bare minimum, you'll need:
  - SQLiteOpenHelper or a DB helper class, and
  - SQLiteDatabase or a reference to the DB itself

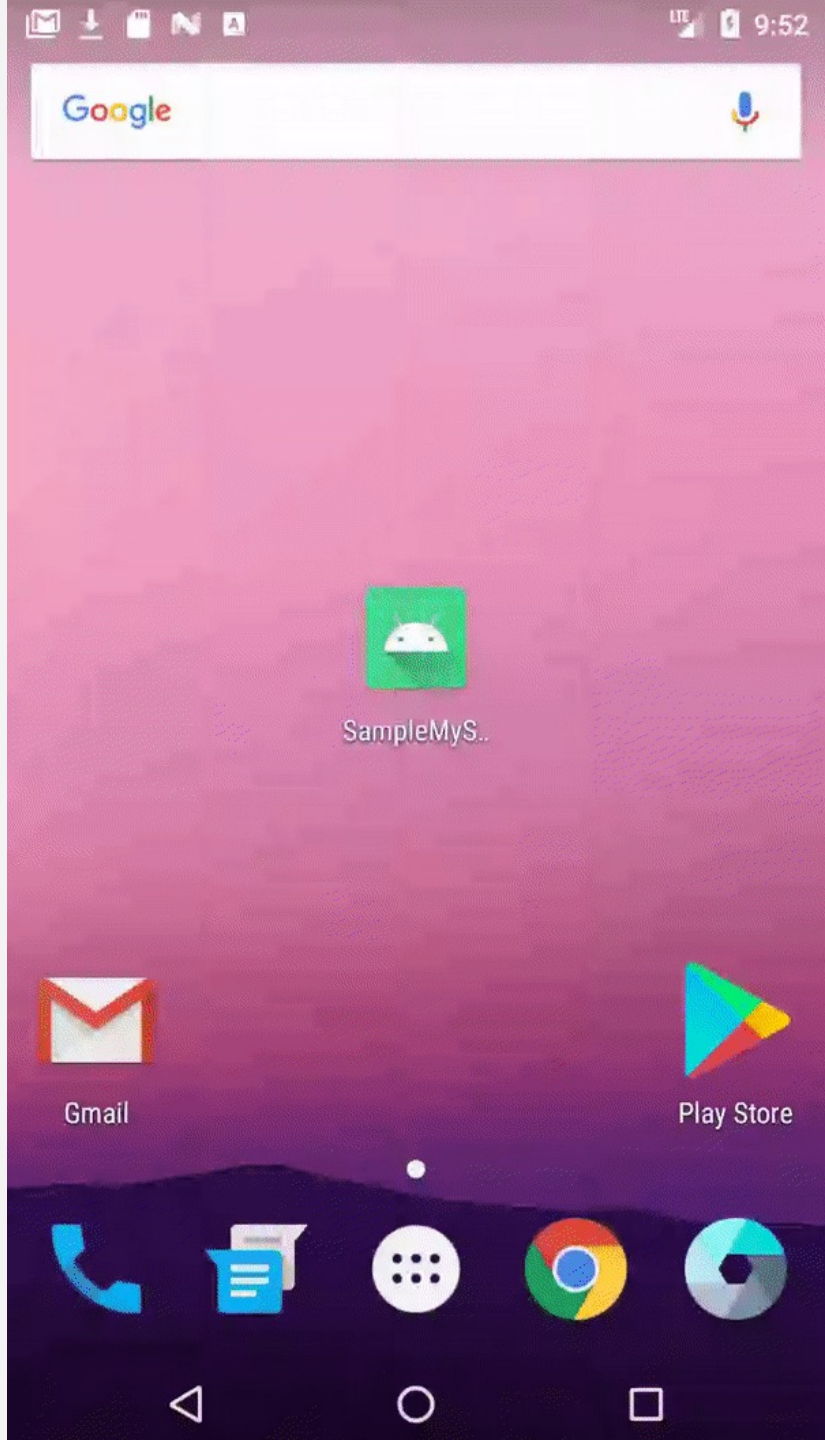
Implementation

- Just like with the RecyclerView lecture, let's define a couple of steps that we can follow:
  1. Define your data model
  2. Define an SQLiteOpenHelper class
     - *Implementing the appropriate constructor and methods and make use of a class to reference common statements, like the table / column names*
  3. Utilize the SQLiteOpenHelper to access SQLiteDatabase
  4. Utilize SQLiteDatabase instance to perform operations

This sorta assumes you have a schema already in mind

1. Defining the Data Model

- Just like with the RecyclerView, we need our Model
- For our case, we'll need variables for
  - *FirstName, LastName, Number, ImageUri*
  - We'll also need to include an *ID* so we know where it is in the DB
- We'll also need the appropriate getters



```java
public class Contact {
    private long id;
    private String lastName, firstName, number, imageUri;

    // Constructor without ID. This isn't exactly advised as you'll have a hard
    // time trying to update the data without the ID reference
    public Contact(String lastName, String firstName, String number, String imageUri) {
        this.lastName = lastName;
        this.firstName = firstName;
        this.number = number;
        this.imageUri = imageUri;
        this.id = null;
    }

    // This is the more appropriate constructor to use because we have a reference
    // of the Contact's id from the DB
    public Contact(String lastName, String firstName, String number, String imageUri, long id) {
        this.lastName = lastName;
        this.firstName = firstName;
        this.number = number;
        this.imageUri = imageUri;
        this.id = id;
    }
    // Getters here!
}
```

```java
public long getId() {
    return id;
}

public String getLastName() {
    return lastName;
}

public String getFirstName() {
    return firstName;
}
```

```java
public String getNumber() {
    return number;
}

public String getImageUri() {
    return imageUri;
}
```

The 2nd constructor is the more appropriate one to use once we have our DB up and running because we'd want to make sure our data can be properly identified.

However, there might be cases where you need a Contact object before it being referenced by the DB or maybe you want your own manner of creating IDs. In that case, the 1st constructor is usable or modifiable.

2. Define the SQLiteOpenHelper Class

- SQLiteOpenHelper is a helper class that contains APIs for managing our databases
  - Used for obtaining a reference of our DB
  - With an instance of a SQLiteOpenHelper, we can use…
    - getWriteableDatabase()
    - getReadableDatabase()

Both actually return the same object – WritableDatabase.

If the disk is full, getWriteableDatabase() may throw an error and getReadableDatabase() will return a ReadbleDatabase

While defining the SQLiteOpenHelper Class…

- Create a class to hold all your constants / statements
  - This is actually an optional step, but it does help in organizing your code
- Code provided has private access modifiers because its within the DB Helper

You can define this in whatever way you'd want or even choose not to have it. Still, organizing your code goes a long way ☺

```
private final class DbReferences {
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "my_database.db";

    private static final String
            TABLE_NAME = "contacts",
            _ID = "id",
            COLUMN_NAME_FIRST_NAME = "first_name",
            COLUMN_NAME_LAST_NAME = "last_name",
            COLUMN_NAME_NUMBER = "number",
            COLUMN_NAME_IMAGE_URI = "image_uri";

    private static final String CREATE_TABLE_STATEMENT =
            "CREATE TABLE IF NOT EXISTS " + TABLE_NAME + " (" +
            _ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
            COLUMN_NAME_FIRST_NAME + " TEXT, " +
            COLUMN_NAME_LAST_NAME + " TEXT, " +
            COLUMN_NAME_NUMBER + " TEXT, " +
            COLUMN_NAME_IMAGE_URI + " TEXT)";

    private static final String DROP_TABLE_STATEMENT =
            "DROP TABLE IF EXISTS " + TABLE_NAME;
}
```

2. Define the SQLiteOpenHelper Class

- When creating a SQLiteOpenHelper class, you'll need to implement 3 things:
  - Constructor()
  - onCreate()
  - onUpgrade()

2. Define the SQLiteOpenHelper Class

- When creating a SQLiteOpenHelper class, you'll need to implement 3 things:
  - Constructor()
    - There are actually 3 constructors you can implement; however, common practice is to only require Context and define the rest of the variables

```
public static final int DATABASE_VERSION = 1;
public static final String DATABASE_NAME = "my_database.db";
```

```
public class MyDbHelper extends SQLiteOpenHelper {

  public MyDbHelper(Context context) {
    super(context, DbReferences.DATABASE_NAME, null, DbReferences.DATABASE_VERSION);
  }
```

Summary

Public constructors

`SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version)`

Create a helper object to create, open, and/or manage a database.

`SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version, DatabaseErrorHandler errorHandler)`

Create a helper object to create, open, and/or manage a database.

`SQLiteOpenHelper(Context context, String name, int version, SQLiteDatabase.OpenParams openParams)`

Create a helper object to create, open, and/or manage a database.

2. Define the SQLiteOpenHelper Class

- When creating a SQLiteOpenHelper class, you'll need to implement 3 things:
  - Constructor()
    - There are actually 3 constructors you can implement; however, common practice is to only require Context and define the rest of the variables

```
public static final int DATABASE_VERSION = 1;
public static final String DATABASE_NAME = "my_database.db";

public class MyDbHelper extends SQLiteOpenHelper {

  public MyDbHelper(Context context) {
    super(context, DbReferences.DATABASE_NAME, null, DbReferences.DATABASE_VERSION);
  }
```

Summary

Public constructors

`SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version)`

Create a helper object to create, open, and/or manage a database.

`SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version, DatabaseErrorHandler errorHandler)`

Create a helper object to create, open, and/or manage a database.

`SQLiteOpenHelper(Context context, String name, int version, SQLiteDatabase.OpenParams openParams)`

Create a helper object to create, open, and/or manage a database.

context -> from the application; **name** -> name of DB;
factory -> for creating cursors (can be null); **version** -> current version of the DB

2. Define the SQLiteOpenHelper Class

- When creating a SQLiteOpenHelper class, you'll need to implement 3 things:
  - onCreate()
    - Is just like what you'd expect from the onCreate() of an activity – its where you initialize the DB
    - This is where you'd normally create your Tables and/or perform initial population

For our example, we only need one Table for the Contacts

```
@Override
public void onCreate(SQLiteDatabase sqLiteDatabase) {
    sqLiteDatabase.execSQL(DbReferences.CREATE_TABLE_STATEMENT);
}
```

```
private static final String CREATE_TABLE_STATEMENT =
    "CREATE TABLE IF NOT EXISTS " + TABLE_NAME + " (" +
    _ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
    COLUMN_NAME_FIRST_NAME + " TEXT, " +
    COLUMN_NAME_LAST_NAME + " TEXT, " +
    COLUMN_NAME_NUMBER + " TEXT, " +
    COLUMN_NAME_IMAGE_URI + " TEXT)";
```

2. Define the SQLiteOpenHelper Class

- When creating a SQLiteOpenHelper class, you'll need to implement 3 things:
  - onUpgrade()
    - Is called if the Helper recognizes that there's an existing DB in place that has a different version
    - Changes might occur when you're pushing updates to the application

```
@Override
public void onUpgrade(SQLiteDatabase sqLiteDatabase, int version1, int version2) {
    sqLiteDatabase.execSQL(DbReferences.DROP_TABLE_STATEMENT);
    onCreate(sqLiteDatabase);
}
```

```
private static final String DROP_TABLE_STATEMENT =
        "DROP TABLE IF EXISTS " + TABLE_NAME;
```

You can place whatever logic you have in mind for upgrading, but for our simple app, we're simply dropping the Contact table and called onCreate() again

And that’s it!

We have a bare minimum DB Helper in place ☺

3-4. SQLiteOpenHelper + SQLiteDatabase

- To access the database instance itself, we need to utilize our defined DB Helper

```
this.myDbHelper = new MyDbHelper(this);
SQLiteDatabase database = this.myDbHelper.getReadableDatabase();

database.delete();
database.update();
database.insert();

database.close();
```

Declare your DB Helper

Get a reference of the DB instance

Call the operations you need to perform

Close the DB instance when done

You'll most likely need to reference the API to know what exactly to provide ☺

Example of Inserting

```
SQLiteDatabase database = this.myDbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(DbReferences.COLUMN_NAME_LAST_NAME, c.getLastName());
values.put(DbReferences.COLUMN_NAME_FIRST_NAME, c.getFirstName());
values.put(DbReferences.COLUMN_NAME_NUMBER, c.getNumber());
values.put(DbReferences.COLUMN_NAME_IMAGE_URI, c.getImageUri());

// Insert the new row
// Inserting returns the primary key value of the new row, but
// we can ignore that if we don't need it
database.insert(DbReferences.TABLE_NAME, null, values);

database.close();
```

Get a reference of the DB instance

Utilize a ContentValues object to ready the values.

This of this as like an Extra. It requires a string parameter representing the column name and the value to be inserted.

Perform the insert operation

Close the DB instance when done

Example of Querying

```
SQLiteDatabase database = this.myDbHelper.getReadableDatabase();
Cursor c = database.query(
        DbReferences.TABLE_NAME,
        null, null, null, null, null,
        DbReferences.COLUMN_NAME_LAST_NAME + " ASC, " +
        DbReferences.COLUMN_NAME_FIRST_NAME + " ASC",
        null
);
while(c.moveToNext()) {
    this.contacts.add(new Contact(
            c.getString(c.getColumnIndexOrThrow(DbReferences.COLUMN_NAME_LAST_NAME)),
            c.getString(c.getColumnIndexOrThrow(DbReferences.COLUMN_NAME_FIRST_NAME)),
            c.getString(c.getColumnIndexOrThrow(DbReferences.COLUMN_NAME_NUMBER)),
            c.getString(c.getColumnIndexOrThrow(DbReferences.COLUMN_NAME_IMAGE_URI)),
            c.getLong(c.getColumnIndexOrThrow(DbReferences._ID))
    ));
}
c.close();
database.close();
```

Get a reference of the DB instance

Perform a query passing:

- table
- columns
- selection
- selectionArgs
- groupBy
- having
- orderBy
- limit

Move through each object in the cursor

Close the DB instance and the cursor when done

Questions so far?

Design Considerations

Considerations

- There are several design considerations that need to be kept in mind when designing how your components interact
  - This is mostly driven by concurrency issues
- The three we'll mention here are:
  - Encapsulating operations within SQLiteOpenHelper
  - Singleton Pattern for the SQLiteOpenHelper
  - Execution of operations off the main thread

Encapsulating operations

- Instead of relying on other classes to handle operations, we can allow our SQLiteOpenHelper to perform the operations
  - This can help in reducing redundant code ☺

In SQLiteOpenHelper…

```
public synchronized void insertContact(Contact c) {
   SQLiteDatabase database = this.getWritableDatabase();
   ContentValues values = new ContentValues();
   values.put(DbReferences.COLUMN_NAME_LAST_NAME, c.getLastName());
   values.put(DbReferences.COLUMN_NAME_FIRST_NAME, c.getFirstName());
   values.put(DbReferences.COLUMN_NAME_NUMBER, c.getNumber());
   values.put(DbReferences.COLUMN_NAME_IMAGE_URI, c.getImageUri());
   database.insert(DbReferences.TABLE_NAME, null, values);
   database.close();
}
```

In Activity…

```
myDbHelper.insertContact(new Contact(
      lastNameEtv.getText().toString(),
      firstNameEtv.getText().toString(),
      numberEtv.getText().toString(),
      imageUri.toString()
));
```

Singleton Pattern for SQLiteOpenHelper

```
public class MyDbHelper extends SQLiteOpenHelper {
   public static MyDbHelper instance = null;

   public MyDbHelper(Context context) {
      super(context, DbReferences.DATABASE_NAME, null, DbReferences.DATABASE_VERSION);
   }

   public static synchronized MyDbHelper getInstance(Context context) {
      if (instance == null) {
         instance = new MyDbHelper(context.getApplicationContext());
      }

      return instance;
   }
}
```

We might want to limit all calling processes to a ***single instance*** of the SQLiteOpenHelper.

In Activity…

```
myDbHelper = MyDbHelper.getInstance(MainActivity.this);
contacts = myDbHelper.getAllContactsDefault();
```

To do so, we implement a getInstance() method that will initialize a single instance of MyDbHelper. If instantiation has already taken place, pass the same instance to the requesting process. You can also synchronize the method to ensure processes finish execution before other processes have access to it.

Execution off of Main Thread

- We haven't discussed concurrency or process management yet, but we need to realize the DB operations can take long
  - If the Main Thread is blocked for around 5 secs, an "app is waiting" prompt will be shown to the user
  - This is how the Android OS works
- Hence, we should look to perform DB operations in a process outside of the Main Thread
  - Main/UI Thread should be responsible for the user interaction

Execution off of Main Thread

- To get around this, we can use an ExecutorService
  - Don't worry about specifics as we'll tackle them when we reach Process Management ☺

In Activity…

```
private ExecutorService executorService = Executors.newSingleThreadExecutor();
…

executorService.execute(new Runnable() {
    @Override
    public void run() {
        myDbHelper = MyDbHelper.getInstance(AddContactActivity.this);
        myDbHelper.insertContact(new Contact(
                lastNameEtv.getText().toString(),
                firstNameEtv.getText().toString(),
                numberEtv.getText().toString(),
                imageUri.toString()
        ));
    }
});
```

Pass in a Runnable object with your DB operations inside

Any questions?

Summary

- Today, we discussed about SQLite as a means to store vertically scaling data
- We learned about the SQLiteOpenHelper and the SQLiteDatabase classes and how we can come up with an SQLite implementation
- We also discussed a couple of design considerations that mostly revolve around concurrency issues

Parting Note

- Android actually recommends to use Room for local database
  - Room builds on top of SQLite and provides for a lot of abstraction – leading to more efficient code
  - However, the overall architecture needed is a little more… complicated than what was discussed here
    - Implementation can be simplified however, so you can look into Room if you're interested to learn more ☺

Thanks everyone!

Video sauce: https://fb.watch/7hDPb9yuep/