KCL Tech Build X: Android

Lecture Five: Preferences and Databases

- Storing Data on a User's Device
- Shared Preferences
- Databases

Storing Data on a User's Device

There are limitless reasons as to why you'd want to store data on a user's device: remembering their preferences, saving their progress in a game, storing content for your app, etc. In our case, we'll be storing the user's list of tasks on the device. This data **persists** even when the app closes or the device reboots.

There are three main methods you can use to store data on a user's device:

- **Plain files**, on the internal or external (SD card) memory - we won't be studying or using these.
- **Shared preferences** - these allow easy storage and access of key/value pairs of information.
- **Databases** - these allow for storage of more structured or complex data.

Shared Preferences

Android shared preferences are designed as a way to **store and share** a user's settings and preferences throughout your app. However, there's no limit on what you can *actually* use this tool for. You can easily store any kind of **key/value** data. Here are some examples of data that might be stored:

Key	Value
`user_highscore`	`42`
`welcome_tour_finished`	`true`
`notification_alert_sound`	`"bells"`

To **write an item** into shared preferences, you need to get an instance of the preference manager. This can be done in a few lines of code, as follows:

```
// get hold of the default shared preferences instance
SharedPreferences prefs =
    PreferenceManager.getDefaultSharedPreferences(getApplicationContext());

// start editing the data
SharedPreferences.Editor prefEditor = prefs.edit();

// write in whatever variables we want, as key/values
prefEditor.putInt("user_highscore", 42);
prefEditor.putBoolean("welcome_tour_finished", true);
prefEditor.putString("notification_alert_sound", "bells");

// finish editing
prefEditor.apply();
```

Reading items out of shared preferences is just as easy: you provide the key and a default value, and the method will return the saved value if you've set it before, or the default if you haven't.

```
// get hold of the default shared preferences instance
SharedPreferences prefs =
    PreferenceManager.getDefaultSharedPreferences(getApplicationContext());

// read out whatever variables we want, using the keys
prefs.getInt("user_highscore", 0);                  // returns 42
prefs.getBoolean("welcome_tour_finished", false);   // returns true
prefs.getString("notification_alert_sound", null); // returns "bells"
prefs.getString("some_new_key", "default val");     // returns "default val"
```

Databases

Databases are used to store **organised, structured data**. In a very, very brief nutshell:

- A single **database** is a collection of **tables**.
- A single **table** stores information about **related objects** (e.g. a "*users"* table).
- A **table** has **rows** and **columns**.
    - A **column** is one of the **properties/attributes** of the stored data (e.g. a *"name"* column).
    - A **row** is one **item/record** of the data (e.g. a row for you, a row for me, a row for Alan, etc.).
- A **query** is a **question or command** given to the database (e.g. *"get all the users who are male"*, *"create a new user called Paige"*, etc.).

A lot of common database systems are built around something called **structured query language**, or **SQL**. This is just a language made up of **instructions** to the database to create, read, update and delete records.

Android uses something called **SQLite**, which is (as the name suggests) a lightweight version of SQL, perfect for running on small devices that can be quite low on processing power. We're going to go through creating a database handler, creating a table, inserting records and retrieving (reading) them.

Creating a Database Handler

Just as an adapter helps to do 90% of the work when you use a ListView, there's a magical class in Android used to make working with databases **a lot easier**. It's called `SQLiteOpenHelper` - you can *extend* it to implement your own database functions and let it do all of the "heavy lifting" in the background.

The first step is to create a class extending `SQLiteOpenHelper`, name your database, and declare the version number. All of this is done by creating a **constructor**.

```
public class DbHelper extends SQLiteOpenHelper {

    public static final String DB_NAME = "KclTechTodo";
    public static final int DB_VERSION = 1;

    public DbHelper(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
    }

}
```

After this, you need to specify two vital methods: `onCreate(...)` and `onUpdate(...)`.

- `onCreate(...)` is called when your code tries to access the database but it has never been used before. This is where the database is initially created.
- `onUpdate(...)` is called when your code tries to access the database and it *has* been used before, but the version on the device is out of date. This is where changes to the database can be made.

We're going to use an **open switch statement** to handle our creation and update segments - this will be covered during a live coding session in the lecture.

Creating a Table

We will be using the following table schema to store our tasks. This strays into database theory so you do not need to understand everything about this, as long as you have a rough idea of what's going on.

```
CREATE TABLE Tasks (
    id INTEGER PRIMARY KEY,
    title TEXT,
    notes TEXT,
    due_date INTEGER,
    is_complete INTEGER
);
```

This is an example of SQL: it is an instruction, telling our database exactly how to create a table called `Tasks` for you, with columns called `id`, `title`, etc. To execute this, we can simply call `execSql(...)` on our writeable database object:

```
db.execSql("CREATE TABLE Tasks ( ... );");
```

Creating and Updating Records

You *could* use SQL to insert a record, but there's an easier way in Android that uses a class called `ContentValues`. An object of this type simply stores **key/value pairs of information** about something, almost exactly like a Bundle does (from lecture three).

Once you convert the record you're inserting into a `ContentValues` object (which we'll be doing in live code), you can insert into your database with straightforward code like this:

```
public void saveTask(Task t) {
    SQLiteDatabase db = getWritableDatabase();
    if (db == null) return;

    db.insertWithOnConflict(
        "Tasks",
        null,
        t.getContentValues(),
        SQLiteDatabase.CONFLICT_REPLACE
    );
}
```

This inserts the task into the database, and uses the keyword `CONFLICT_REPLACE` to tell SQLite to replace any record that it might already have with the same ID (allowing you to **create a new record** *or* **update an existing record** in the same method).

Reading Records

When it comes to pulling records back out of the database, you need to start playing with SQL. Once again, because we're not studying database theory, you only need a basic understanding of the following queries:

To read all uncomplete tasks, sorted by soonest due date first:

```
SELECT * FROM Tasks WHERE is_complete = 0 ORDER BY due_date ASC;
```

To read a specific task, using it's ID:

```
SELECT * FROM Tasks WHERE id = ?;
```

(The query helper will replace the `?` with the ID you specify.)

Either of these queries can be executed by calling `rawQuery(...)` on a readable database object:

```
SQLiteDatabase db = getReadableDatabase();

Cursor allTasks = db.rawQuery("... WHERE is_complete = 0 ...");

Cursor task22 = db.rawQuery("... WHERE id = ? ...", new String[]{ "22" });
```

Both of these methods return a `Cursor` object. A `Cursor` is a collection of records from a database that can be iterated (looped) over easily. To loop over a `Cursor`, the following pattern is used (this will all be explained during the lecture):

```
Cursor allTasks = db.rawQuery(...);
if (allTasks != null && allTasks.moveToFirst()) {
    do {
        // do something with a single record
    } while (allTasks.moveToNext());
}
```

A cursor is **one object** that can represent **any number of results** (from zero to thousands or more).

This works because the cursor only "looks at" **one result at a time**, and it maintains an internal **pointer** to remember which result is currently being "looked at". You can move the pointer back and forth, and when it is referring to a valid result, you can use methods like `getInt()` and `getString()` to return properties of the current record is it looking at.

`moveToFirst()` will move the pointer to the first record; it returns `false` if there isn't a "first" record, so the loop can be skipped. We then use `moveToNext()` to keep stepping through the results; it returns `false` when there isn't a "next" record (i.e. the end of the results), so we can break out of the loop.