Mobile Programming

Course 4

Data storage

Qiong LIU

qiong.liu@cyu.fr

CY Tech, CY Cergy Paris University

2024 - 2025

Database Operations

Database is everywhere:

- Data Storage: Save user data, app settings, and application states.
- Resource allocation: Efficiently query and manipulate data for faster app performance.
- Offline Support: Ensure app functionality even without an internet connection.
- pre-loaded: Save user preferences and custom settings
- ...

What We'll Learn Today

Database design for Android

1. Shared-preferences
   - *When you just need to store a few values quickly and easily.*
2. SQLite in Android
   - *Structured, searchable local data — but needs template code.*

Method 1: Shared Preferences

Method 1: Shared Preferences

Properties:

- A simple way to store small amounts of data
- Stores data as key-value pairs, such as:
  - Key: `"username"`
  - Value: `"JohnDoe"`
- Supports data types like:
  - `boolean`, `int`, `float`, `long`, `string`
- Data is saved in a private XML file inside the app's storage
- Data persists even when the app is closed

Use Case Example

- Remembering a user's dark mode preference
- Storing last login timestamp
- User settings

Method 1: Shared Preferences

Does not support full CRUD operations like a database

Required Imports:

- `import android.content.SharedPreferences;`
- `import android.content.Context;`

Operation	**Method**	**Description**
Create / Update	`putString()`, `putInt()`	Insert or update a key-value pair
Read	`getString()`, `getBoolean()`	Retrieve a value by key
Delete	`remove(key)`	Remove a specific key-value pair
Delete All	`clear()`	Clear all stored data
Save Changes	`apply()` or `commit()`	Commit changes (apply is async)

Table: Typical SharedPreferences Operations

Method 1: Shared Preferences

Save data:

- Use `getSharedPreferences()` to get an instance
- Call `edit()` → put data → `apply()` or `commit()`

Code Example

```
// Get the SharedPreferences object named "myPrefs",
   MODE_PRIVATE means only this app can access the data
SharedPreferences prefs = getSharedPreferences("myPrefs",
   MODE_PRIVATE);

// Get an Editor object to make changes to SharedPreferences
SharedPreferences.Editor editor = prefs.edit();

editor.putString("username", "Alice");
editor.putBoolean("loggedIn", true);
editor.apply();
```

`apply()` saves asynchronously; `commit()` is synchronous.

Method 1: Shared Preferences

Read data:

- Use key to retrieve the saved value
- Provide default value in case key is missing

Code Example

```
// Read data
SharedPreferences prefs = getSharedPreferences("myPrefs",
    MODE_PRIVATE);
String name = prefs.getString("username", "Guest");
boolean loggedIn = prefs.getBoolean("loggedIn", false);
```

Always set a default value to avoid null or crashes.

Method 1: Shared Preferences

Delete data:

- Use `remove(key)` to delete a specific value
- Use `clear()` to delete all key-value pairs
- Always call `apply()` or `commit()` to save changes

Code Example

```
// Delete a specific key
SharedPreferences prefs = getSharedPreferences("myPrefs",
    MODE_PRIVATE);
SharedPreferences.Editor editor = prefs.edit();
editor.remove("username"); // removes the key "username"
editor.apply(); // or editor.commit()

// Delete all stored data
editor.clear(); // removes all key-value pairs
editor.apply(); // don't forget to apply the changes
```

Use `clear()` with caution – it wipes everything in the file.

Method 1: Shared Preferences

If you callgetSharedPreferences() with different names, you're working with different files. It is automically stored in /data/data/your.package.name/shared_prefs/myPrefs.xml

```
SharedPreferences prefs1 = getSharedPreferences("loginPrefs"
    , MODE_PRIVATE);
SharedPreferences prefs2 = getSharedPreferences("themePrefs"
    , MODE_PRIVATE);
```

Method 1: Shared Preferences

Ways to organize your preferences:

- Option 1: Use different files for different modules
  - `getSharedPreferences("login_prefs", MODE_PRIVATE)`
  - `getSharedPreferences("profile_prefs", MODE_PRIVATE)`
- Option 2: Use one file, prefix keys
  - `"login_username"`, `"login_status"`
  - `"profile_theme"`, `"profile_fontSize"`
  - Just make sure all keys are unique within the same file

SharedPreferences are global across activities

Organize your keys clearly to avoid conflicts and bugs.

Method 2: SQLite

SQLite

SQLite:

- Embedded relational database (no server needed)
- Uses SQL syntax for queries (CRUD operations)
- Persistent: stored inside app's private file directory
- Good for: structured data (contacts, courses, logs, etc.)

Example: Student Table

id	name	age
1	Alice	20
2	Bob	22

This structure is the foundation for how data is stored in SQLite.

SQLite Storage Types (with Examples)

- `NULL` – A missing value
  - *Example:* ‘age = NULL‘ (unknown age)
- `INTEGER` – A signed integer (1 to 8 bytes depending on size)
  - *Example:* ‘user_id = 1001‘, ‘age = 25‘
- `REAL` – A floating point number (8-byte IEEE 754)
  - *Example:* ‘price = 19.99‘, ‘latitude = 48.8566‘
- `TEXT` – A text string stored as UTF-8 or UTF-16
- `BLOB` – A binary object, stored exactly as input
  - *Example:* profile image, audio file, or PDF stored as bytes

How to Use SQLite in Android

Steps to use SQLite in Android:

1. Create a custom class that extends `SQLiteOpenHelper`
2. Override `onCreate()` to define the schema (tables)
3. Override `onUpgrade()` to handle schema changes
4. Use `SQLiteDatabase` methods to insert, query, update, delete

All data is saved in a local .db file on the device

SQLiteOpenHelper helps manage database creation and version control.

Fragment3: Course Management

What this Fragment does:

- Let students enter course info: `name, teacher, time`
- Save data into SQLite database
- Show all saved courses in a list (ListView)
- Long-click a course to delete it

This demonstrates: SQLite insert, query, delete with a simple UI.

Pre: Create a Database

Create a subclass of SQLiteOpenHelper and override `onCreate()`:

```java
public class CourseDatabaseHelper extends SQLiteOpenHelper {
    // version type
    private static final String DATABASE_NAME = "course.db";
    private static final int DATABASE_VERSION = 1;
    // list name
    public static final String TABLE_NAME = "courses";
    public static final String COLUMN_ID = "id";
    ...
    // creat SQL
    private static final String TABLE_CREATE =
            "CREATE TABLE " + TABLE_NAME + " (" +
                    COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                    COLUMN_COURSE_NAME + " TEXT, " +
                    COLUMN_TEACHER + " TEXT, " +
                    COLUMN_TIME_SLOT + " TEXT" +
                    ");";
    public CourseDatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(TABLE_CREATE);
    }
}
```

Pre: Action queries

Every time you write to the database

- Grab an instance of your SQLiteOpenHelper
- Call `getWritableDatabase()`
- This returns a SQLiteDatabase object that represents the database and provides methods for SQLite operations.
- When your app is destroyed, close database by calling `close()`

```
MyDatabaseHelper helper = new MyDatabaseHelper();
SQLiteDatabase db = helper.getWritableDatabase();
...
db.insert(...); // or update or delete
...
db.close();
```

Step 1: Setup UI and DB Helper

```
View view = inflater.inflate(R.layout.fragment_3, container,
     false);

// Initialize DB helper
dbHelper = new CourseDatabaseHelper(requireContext());

// Bind EditTexts and Button
editCourseName = view.findViewById(R.id.edit_course_name);
editTeacher = view.findViewById(R.id.edit_teacher);
editTimeSlot = view.findViewById(R.id.edit_time_slot);
buttonAdd = view.findViewById(R.id.button_add_course);
```

Use SQLiteOpenHelper to manage DB. Inflate layout, link input fields.

Step 2: Insert Data into SQLite

```
buttonAdd.setOnClickListener(v -> insertCourse());

private void insertCourse() {
  SQLiteDatabase db = dbHelper.getWritableDatabase();
  db.execSQL("INSERT INTO courses (course_name, teacher,
      time_slot) VALUES (?, ?, ?)",
      new Object[]{courseName, teacher, timeSlot});
}
```

- Get input text
- Validate input
- Use execSQL to insert row

Step 3: Read Data and Display

```
Cursor cursor = db.rawQuery("SELECT * FROM courses", null);
if (cursor.moveToFirst()) {
  do {
    String info = courseName + " (" + teacher + ") - " +
        time;
    courseList.add(info);
  } while (cursor.moveToNext());
}
```

ListView shows courseList via ArrayAdapter. *Read all rows using Cursor and format into strings.*

Step 4: Delete Course on Long Press

```
listView.setOnItemLongClickListener((parent, view, pos, id)
    -> {
  String item = courseList.get(pos);
  String courseName = item.split(" \\(")[0];

  db.delete("courses", "course_name = ?", new String[]{
      courseName});
});
```

- Extract course name from display string
- Confirm delete (use AlertDialog)
- Use `db.delete()` to remove

Summary: What We Learned

SQLite with UI in Android:

- `SQLiteOpenHelper` manages DB lifecycle
- Use SQL for **INSERT**, **SELECT**, **DELETE**
- Display data with `ListView` and `ArrayAdapter`
- Cursor reads rows; use AlertDialog for user interaction

Next: update features, or use RecyclerView and DAO.