Chapter 5
Saving Data

Databases in android

- Android uses SQLite to store Relational DB data.

- SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine
  - It requires no configuration and stores information in ordinary disk files

- If your App uses an SQLite DB, it will be saved in the folder: /data/APP_NAME/databases/FILENAME

Databases in android

- The APIs you'll need to use a database on Android are available in the **android.database.sqlite** package.

- SQLite supports the following data types: TEXT, INTEGER and REAL

Databases in android

- Assume we will save some info about users in a table called **contacts**

THREE steps

1. Create an object/class that describes a tuple in your table with methods to read and write values

2. Create a **DB helper class** containing all CRUD operations (Create, Read, Update and Delete Tables)

3. Use the two steps above in our activity class

Step 1: Creating the class

- It's a good idea to create a class representing a tuple (row) in any table we have.
- In our case, each tuple denotes a contact for a person. Other tables might have different types of objects.
- The created class will have a constructor and setter and getter methods.

Step 1: Creating the class

```
public class Contact {
    //private variables
    int _id;
    String _name;
    String _phone_number;

    // Empty constructor
    public Contact(){ }

    // constructor 1 with id
    public Contact(int id, String name, String _phone_number){
        this._id = id;
        this._name = name;
        this._phone_number = _phone_number;
    }

    // constructor 2 without id
    public Contact(String name, String _phone_number){
        this._name = name;
        this._phone_number = _phone_number;}
```

Step 1: Creating the class

```
     // getting ID
    public int getID(){
        return this._id;
    }


    // setting id
    public void setID(int id){
        this._id = id;
    }
    // getting name
    public String getName(){
        return this._name;
    }

    // setting name
    public void setName(String name){
        this._name = name;
    }

    // getting phone number
    public String getPhoneNumber(){
        return this._phone_number;
    }

    // setting phone number
    public void setPhoneNumber(String phone_number){
        this._phone_number = phone_number;
    }
}
```

Step 2: DB helper or handler class

- We need to write our own class to handle all database CRUD (Create, Read, Update and Delete) operations.

- Create a new class in your project src directory and name it as DatabaseHandler.java
  - Extend your DatabaseHandler.java class from **SQLiteOpenHelper**:

```
public class DatabaseHandler extends SQLiteOpenHelper
{

}
```

Step 2: DB helper or handler class

- After extending your class from SQLiteOpenHelper you need to override two methods:

  - **onCreate**( ) – This is where we need to write create table statements. This is called when database is created.

  - **onUpgrade**( ) – This method is called when database is upgraded like modifying the table structure, adding constraints to database etc.,

SQLiteOpenHelper class

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 class

Public methods	
`void`	`close()` Close any open database object.
`String`	`getDatabaseName()` Return the name of the SQLite database being opened, as given to the constructor.
`SQLiteDatabase`	`getReadableDatabase()` Create and/or open a database.
`SQLiteDatabase`	`getWritableDatabase()` Create and/or open a database that will be used for reading and writing.
`void`	`onConfigure(SQLiteDatabase db)` Called when the database connection is being configured, to enable features such as write-ahead logging or foreign key support.
`abstract void`	`onCreate(SQLiteDatabase db)` Called when the database is created for the first time.
`void`	`onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion)` Called when the database needs to be downgraded.
`void`	`onOpen(SQLiteDatabase db)` Called when the database has been opened.
`abstract void`	`onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)` Called when the database needs to be upgraded.

Step 2: DatabaseHandler class

```
public class DatabaseHandler extends SQLiteOpenHelper {

    // Database Version
    private static final int DATABASE_VERSION = 1;

    // Database Name
    private static final String DATABASE_NAME =
"contactsManager";

    // Contacts table name
    private static final String TABLE_CONTACTS = "contacts";

    // Contacts Table Columns names
    private static final String KEY_ID = "id";
    private static final String KEY_NAME = "name";
    private static final String KEY_PH_NO = "phone_number";

    public DatabaseHandler(Context c) {
        super(c, DATABASE_NAME, null, DATABASE_VERSION);
    }
```

Step 2: DatabaseHandler class

```
// Creating Tables
    @Override
    public void onCreate(SQLiteDatabase db) {

        String CREATE_CONTACTS_TABLE =
"CREATE TABLE " + TABLE_CONTACTS + "("+ KEY_ID + " INTEGER PRIMARY KEY," +
 KEY_NAME + " TEXT," + KEY_PH_NO + " TEXT" + ")";

        db.execSQL(CREATE_CONTACTS_TABLE);
    }


    // Upgrading database
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // Drop older table if existed
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_CONTACTS);

        // Create tables again
        onCreate(db);
    }
```

In SQLITE, PRIMARY KEY has AUTO INCREMENT by default!

SQLiteDatabase class

- SQLiteDatabase has methods to create, delete, execute SQL commands, and perform other common database management tasks (~65 methods!).

- http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html

SQLiteDatabase class

```
public void execSQL (String sql)
```

Execute a single SQL statement that is NOT a SELECT or any other SQL statement that returns data.

It has no means to return any data (such as the number of affected rows). Instead, you're encouraged to use `insert(String, String, ContentValues)`, `update(String, ContentValues, String, String[])`, et al, when possible.

Parameters

sql the SQL statement to be executed. Multiple statements separated by semicolons are not supported.

Throws

SQLException if the SQL string is invalid

SQLiteDatabase class

```
public long insert (String table, String
  nullColumnHack, ContentValues values)
```

Convenience method for inserting a row into the database.

Parameters

table	the table to insert the row into
nullColumnHack	optional; may be `null`. SQL doesn't allow inserting a completely empty row without naming at least one column name. If your provided `values` is empty, no column names are known and an empty row can't be inserted. If not set to null, the `nullColumnHack` parameter provides the name of nullable column name to explicitly insert a NULL into in the case where your `values` is empty.
values	this map contains the initial column values for the row. The keys should be the column names and the values the column values

Returns

the row ID of the newly inserted row, or -1 if an error occurred

the `ContentValues` is a set of key-value pairs
the key represents the column for the table
the value is the value to be inserted in that column.

```
Example: values.put("calendar_id", 1);
```

SQLiteDatabase class

```
public Cursor query (String table, String[] columns, String
selection, String[] selectionArgs, String groupBy, String having,
String orderBy)
```

Parameters

table	The table name to compile the query against.
columns	A list of which columns to return. Passing null will return all columns, which is discouraged to prevent reading data from storage that isn't going to be used.
selection	A filter declaring which rows to return, formatted as an SQL WHERE clause (excluding the WHERE itself). Passing null will return all rows for the given table.
selectionArgs	You may include ?s in selection, which will be replaced by the values from selectionArgs, in order that they appear in the selection. The values will be bound as Strings.
groupBy	A filter declaring how to group rows, formatted as an SQL GROUP BY clause (excluding the GROUP BY itself). Passing null will cause the rows to not be grouped.
having	A filter declare which row groups to include in the cursor, if row grouping is being used, formatted as an SQL HAVING clause (excluding the HAVING itself). Passing null will cause all row groups to be included, and is required when row grouping is not being used.
orderBy	How to order the rows, formatted as an SQL ORDER BY clause (excluding the ORDER BY itself). Passing null will use the default sort order, which may be unordered.

Returns

A `Cursor` object, which is positioned before the first entry. Note that `Cursor`s are not synchronized, see the documentation for more details.

SQLiteDatabase class

`rawQuery(String sql, String[] selectionArgs)`

Runs the provided SQL and returns a `Cursor` over the result set.

Parameters

sql	the SQL query. The SQL string must not be ; terminated
selectionArgs	You may include ?s in where clause in the query, which will be replaced by the values from selectionArgs. The values will be bound as Strings.

Returns

A `Cursor` object, which is positioned before the first entry. Note that `Cursor`s are not synchronized, see the documentation for more details.

SQLiteDatabase class

```
public int update (String table, ContentValues
values, String whereClause, String[] whereArgs)
```

Convenience method for updating rows in the database.

Parameters

table	the table to update in
values	a map from column names to new column values. null is a valid value that will be translated to NULL.
whereClause	the optional WHERE clause to apply when updating. Passing null will update all rows.
whereArgs	You may include ?s in the where clause, which will be replaced by the values from whereArgs. The values will be bound as Strings.

Returns

the number of rows affected

Step 2: DatabaseHandler class

```
// Adding new contact

    void addContact(Contact c) {

        SQLiteDatabase db = this.getWritableDatabase();

        ContentValues values = new ContentValues();
        values.put(KEY_NAME, c.getName());
        values.put(KEY_PH_NO, c.getPhoneNumber());

        // Inserting Row
        db.insert(TABLE_CONTACTS, null, values);
        db.close(); // Closing database connection
    }
```

Step 2: DatabaseHandler class

```
// Getting single contact
Contact getContact(int id) {
    SQLiteDatabase db = this.getReadableDatabase();

    Cursor cursor = db.query(TABLE_CONTACTS, new String[] { KEY_ID,
            KEY_NAME, KEY_PH_NO }, KEY_ID + "=?",
            new String[] { String.valueOf(id) }, null, null, null, null);
    if (cursor != null)
    {
        cursor.moveToFirst();

    Contact c = new Contact(Integer.parseInt(cursor.getString(0)),
            cursor.getString(1), cursor.getString(2));

            // return contact
            return c;
    }
    db.close();
    return null;
}
```

Step 2: DatabaseHandler class

```
// Getting single contact
    List<Contact> getAllContacts() {
        SQLiteDatabase db = this.getReadableDatabase();
        String query="SELECT * FROM "+TABLE_CONTACTS;
        Cursor cursor = db.rawQuery(query, null);

        List<Contact> contacts = new ArrayList<Contact>();

       if(cursor.moveToFirst()){
       do{
           Contact c = new Contact(Integer.parseInt(cursor.getString(0)),
                cursor.getString(1), cursor.getString(2));

           contacts.add(c);

         }while(cursor.moveToNext());
        }
        db.close();
        // return contact
        return contacts;
    }
```

Until now we have:

Contact class

variables

_Id
_Name
_Phone_number

Constructors

Contact()
Contact(x,y,z)
Contact(y,z)

Methods

getID()
getName()
getPhoneNumber()
setID(x)
setName(x)
setPhoneNumber(x)

DatabaseHandler extends SQLiteOpenHelper

variables

Constructors

DatabaseHandler()

Mandatory metnods

onCreate()
onUpgrade()

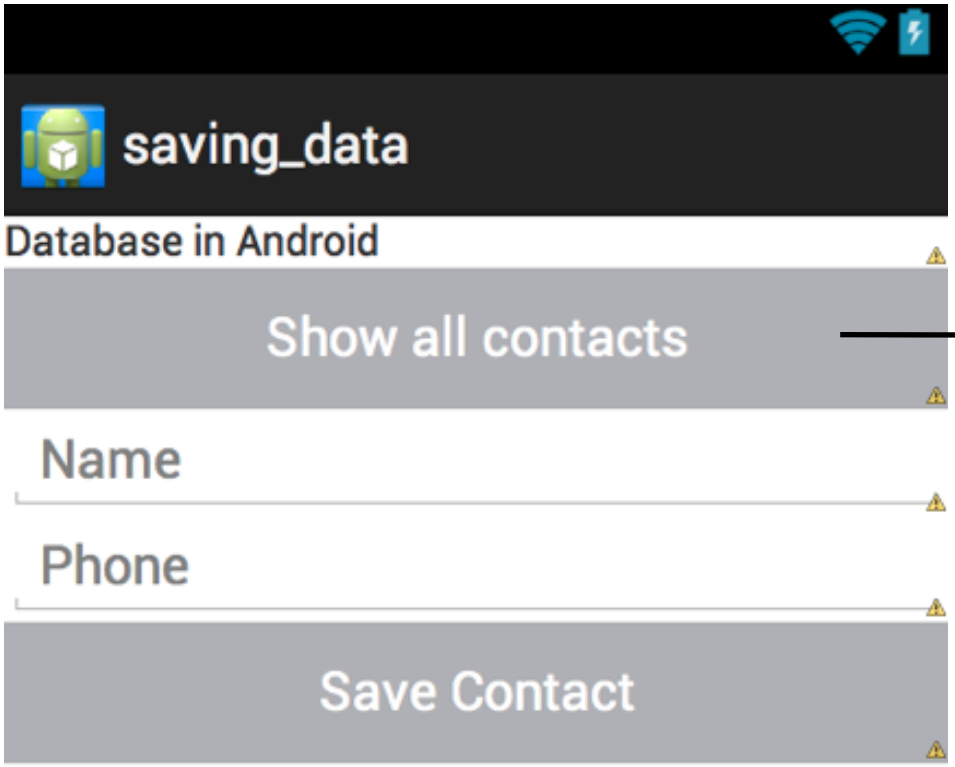
User defined CRUD methods

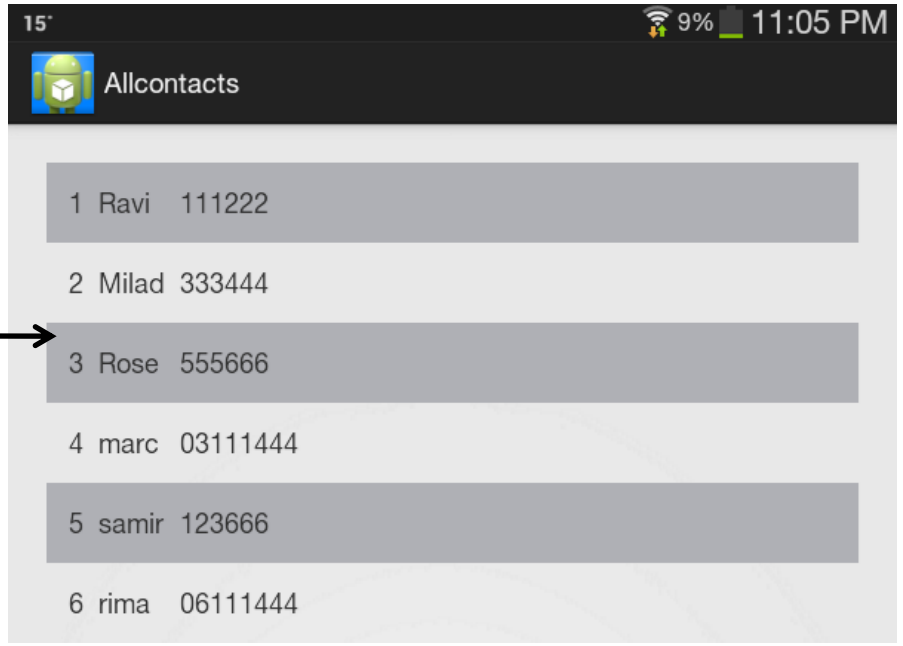
Addcontact, delete, Getall,…
All use the insert, Query, rawQuery, Execsql…

Activity

?

Let's do this!





Allcontacts

MainActivity

XML of main activity

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   tools:context="com.example.saving_data.MainActivity" >

   <TextView
      android:id="@+id/textView1"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"

      android:text="Database in Android" />

   <Button
      android:id="@+id/displayall"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:background="#AFB0B5"
      android:onClick="displayAll"
      android:text="Show all contacts"
      android:textColor="#FFFFFF" />
```

```
<EditText
      android:id="@+id/namefield"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:hint="Name"
      android:inputType="text" />

   <EditText
      android:id="@+id/phonefield"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:hint="Phone"
      android:inputType="phone" />

   <Button
      android:id="@+id/saveContact"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:background="#AFB0B5"
      android:onClick="saveContact"
      android:text="Save Contact"
      android:textColor="#FFFFFF" />

</LinearLayout>
```

Step 3: Main activity SRC code

```
public class MainActivity extends Activity {

    DatabaseHandler db;

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        db = new DatabaseHandler (this);}

public void displayAll(View v){
Intent i = new Intent(this, Allcontacts.class);
        startActivity(i);   }

public void saveContact(View v){

        EditText e1 = (EditText) findViewById(R.id.namefield);
        EditText e2 = (EditText) findViewById(R.id.phonefield);
        String s1 = e1.getText().toString();
        String s2 = e2.getText().toString();
        Contact newContact = new Contact(s1,s2);
       db.addContact(newContact);} }
```

XML of Allcontacts Activity

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
tools:context="com.example.saving_data.Allcontacts" >

    <TableLayout
        android:id="@+id/mytable"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" >

     </TableLayout>



</RelativeLayout>
```

SRC code for Allcontacts Activity

```
public class Allcontacts extends Activity {
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_allcontacts);

                // Get an instance of the Database Handler:

                DatabaseHandler db= new DatabaseHandler(this);

                // execute the method getAllContacts to get all of them into the list called
contacts:

    List<Contact> contacts = db.getAllContacts();
```

Fill the table programmatically from java by looping over all contacts, adding rows and columns!

Android to MySQL

- Assume having a MySQL DB named "db1forandroid" containing the table "tbl_user" with 3 fields (id, firstName, lastName)

- Create a php script named "Connection.php" which will setup the connection to "db1forandroid"

```
<?php
$hostname ="localhost";
$database ="db1forandroid";
$username ="root";
$password ="";

$link = mysqli_connect($hostname, $username, $password) or
trigger_error(mysql_error(), E_USER_ERROR);

mysqli_select_db($link, $database);

?>
```

Android to MySQL

- Create a second php script to be called by your Android app
  - Setup the connection with the DB
  - Receive the user id: return the full name from “tbl_user” if this id exists and otherwise return “N”.

```
<?php
require_once('Connection.php');

$id = $_GET['id'];
$query_search = "select * from tbl_user where id = '".$id."'";
$query_exec = mysqli_query($link, $query_search);
$rows = mysqli_num_rows($query_exec);

if($rows > 0) {
     while(@$row=mysqli_fetch_array($query_exec)){
            echo $row['firstName'] . " " . $row['lastName'];
     }
}
else  {  echo "N"; }
?>
```

Android to MySQL

- Create a simple Android app where the main activity contains an EditText for the user id and a Button for search.
- When the Button is clicked, send the id and wait for the full name
- Use an **HttpURLConnection** object to run the php script on the server.
    - ✓ Should be done in a separate thread (if not your app will be killed by the OS)
- Receive the answer using InputStream.

```
try{
    URL url = new URL("http://serverIP/android/getUser.php?id="+id);
    HttpURLConnection connection =
                        (HttpURLConnection)url.openConnection();

    connection.setRequestMethod("GET");
    connection.connect();

    InputStream inputStream = connection.getInputStream();

    BufferedReader rd = new BufferedReader(new
                        InputStreamReader(inputStream));
     String line = "";
     String answer = "";
     while ((line = rd.readLine()) != null) {
         answer += line;
     }
     System.out.println("User name = " + answer);

}catch(Exception e)
{
     System.out.println(e.toString());
}
```

Full example: **AndroidToMySQL** app