Android Developer Fundamentals

Storing Data

Lesson 10

10.2 SQLite Database

Contents

- SQLite database
- Cursors
- Content Values
- Implementing SQLite
- Backups

$\left\{\right.$

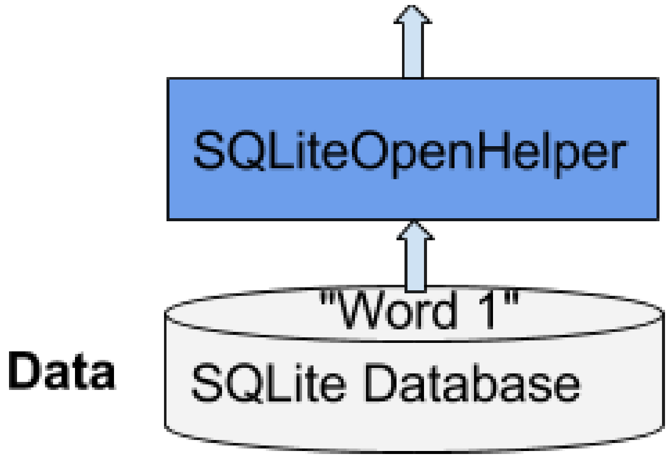
1. Data model
2. Subclass Open Helper
3. Query
4. Insert, Delete, Update, Count
5. Instantiate Open Helper
6. Work with database

SQLite Database

Using SQLite database

- Versatile and straightforward to implement
- Structured data that you need to store persistently
- Access, search, and change data frequently
- Primary storage for user or app data
- Cache and make available data fetched from the cloud
- Data can be represented as rows and columns

Components of SQLite database



SQLiteOpenHelper

SQLite database represented as an SQLiteDatabase object
all interactions with database through SQLiteOpenHelper

- Executes your requests
- Manages your database
- Separates data and interaction from app
- Keeps complex apps manageable

Cursors

Cursors

- Data type commonly used for results of queries
- Pointer into a row of structured data ...
- … think of it as an array of rows
- Cursor class provides methods for moving cursor and getting data
- SQLiteDatabase always presents results as Cursor

Cursor subclasses

- SQLiteCursor exposes results from a query on a SQLiteDatabase
- MatrixCursor is a mutable cursor implementation backed by an array of Objects that automatically expands internal capacity as needed

Cursor common operations

- getCount()—number of rows in cursor
- getColumnNames()—string array with column names
- getPosition()—current position of cursor
- getString(int column), getInt(int column), ...
- moveToFirst(), moveToNext(), ...
- close() releases all resources and invalidates cursor

Processing Cursors

```
// Store results of query in a cursor
Cursor cursor = db.rawQuery(...);
try {
    while (cursor.moveToNext()) {
         // Do something with data
     }
} finally {
    cursor.close();
}
```

Content Values

ContentValues

- An instance of ContentValues
  - Represents one table row
  - Stores data as key-value pairs
  - Key is the name of the column
  - Value is the value for the field
- Used to pass row data between methods

ContentValues

```
ContentValues values = new ContentValues();


// Inserts one row.
// Use a loop to insert multiple rows.
values.put(KEY_WORD, "Android");
values.put(KEY_DEFINITION, "Mobile operating system.");

db.insert(WORD_LIST_TABLE, null, values);
```

Implementing SQLite

You always need to ...

1. Create data model
2. Subclass SQLiteOpenHelper
   a. Create constants for tables
   b. onCreate()—create SQLiteDatabase with tables
   c. onUpgrade(), and optional methods
   d. Implement query(), insert(), delete(), update(), count()
3. In MainActivity, create instance of SQLiteOpenHelper
4. Call methods of SQLiteOpenHelper to work with database

Data Model

Data model

- Class with getters and setters
- One "item" of data (for database, one record or one row)

```
public class WordItem {
    private int mId;
    private String mWord;
    private String mDefinition;
   ...
}
```

Subclass SQLiteOpenHelper

Subclass SQLiteOpenHelper

```
public class WordListOpenHelper extends SQLiteOpenHelper {

    public WordListOpenHelper(Context context) {
       super(context, DATABASE_NAME, null, DATABASE_VERSION);
       Log.d(TAG, "Construct WordListOpenHelper");
    }
}
```

Declare constants for tables

```
private static final int DATABASE_VERSION = 1;
// Has to be 1 first time or app will crash.
private static final String DATABASE_NAME = "wordlist";
private static final String WORD_LIST_TABLE = "word_entries";

// Column names...
public static final String KEY_ID = "_id";
public static final String KEY_WORD = "word";

// ... and a string array of columns.
private static final String[] COLUMNS = {KEY_ID, KEY_WORD};
```

Define query for creating database

- You need a query to create the database
- Customarily defined as a string constant
- SQLite supports the following data types: NULL, TEXT, INTEGER, REAL and BLOB

```
private static final String WORD_LIST_TABLE_CREATE =
            "CREATE TABLE " + WORD_LIST_TABLE + " (" +
                    KEY_ID + " INTEGER PRIMARY KEY, " +
                    // will auto-increment if no value passed
                    KEY_WORD + " TEXT );";
```

onCreate()

```
@Override
public void onCreate(SQLiteDatabase db) { // Creates new database
  // Create the tables
  db.execSQL(WORD_LIST_TABLE_CREATE);
  // Add initial data
  ...
}
```

onUpgrade()

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
                                         int newVersion) {
    // SAVE USER DATA FIRST!!!
    Log.w(WordListOpenHelper.class.getName(),
         "Upgrading database from version " + oldVersion + " to "
         + newVersion + ", which will destroy all old data");
        db.execSQL("DROP TABLE IF EXISTS " + WORD_LIST_TABLE);
        onCreate(db);
    }
```

Optional methods

- onDowngrade()—default rejects downgrade
- onConfigure()—called before onCreate(). Only call methods that configure the parameters of the database connection
- onOpen()

Database Operations

Database operations

- query()
- insert()
- update()
- delete()

Query Method

Executing queries

- implement query() method in open helper class
- query() can take and return any data type that UI needs
- Only support queries that your app needs
- Use database convenience methods for insert, delete, and update

Database methods for executing queries

- SQLiteDatabase.rawQuery()
  Use when data is under your control and supplied only by your app

- SQLiteDatabase.query()
  Use for all other queries

SQLiteDatabase.rawQuery() format

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

- First parameter is SQLite query string
- Second parameter contains the arguments
- Only use if your data is supplied by app and under your full control

SQLiteDatabase.rawQuery() example

```
String query = "SELECT  * FROM " + WORD_LIST_TABLE +
       " ORDER BY " + KEY_WORD + " ASC " +
       "LIMIT " + position + ",1";

cursor = mReadableDB.rawQuery(queryString, null);
```

SQLiteDatabase.query() format

```
Cursor query (boolean distinct,  String table,
              String[] columns, String selection,
              String[] selectionArgs, String groupBy,
              String having, String orderBy,String limit);
```

SQLiteDatabase.query() example

```
String[] columns = new String[]{KEY_WORD};
String where =  KEY_WORD + " LIKE ?";
searchString = "%" + searchString + "%";
String[] whereArgs = new String[]{searchString};
cursor = mReadableDB.query(WORD_LIST_TABLE, columns, where,
         whereArgs, null, null, null);
```

Insert, Delete, Update, Count

insert() format

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

- First argument is the table name.
- Second argument is a `String nullColumnHack`.
    - Workaround that allows you to insert empty rows
    - Use `null`
- Third argument must be a ContentValues with values for the row
- Returns the id of the newly inserted item

insert() example

```
newId = mWritableDB.insert(
    WORD_LIST_TABLE,
    null,
    values);
```

delete() format

```
int delete (String table,
            String whereClause, String[] whereArgs)
```

- First argument is table name
- Second argument is WHERE clause
- Third argument are arguments to WHERE clause

delete() example

```
deleted = mWritableDB.delete(
    WORD_LIST_TABLE,
    KEY_ID + " =? ",
    new String[]{String.valueOf(id)});
```

update() format

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

- First argument is table name
- Second argument must be ContentValues with new values for the row
- Third argument is WHERE clause
- Fourth argument are the arguments to the WHERE clause

update() example

```
ContentValues values = new ContentValues();
values.put(KEY_WORD, word);

mNumberOfRowsUpdated = mWritableDB.update(
                WORD_LIST_TABLE,
                values, // new values to insert
                KEY_ID + " = ?",
                new String[]{String.valueOf(id)});
```

Always!

- Always put database operations in try-catch blocks
- Always validate user input and SQL queries

Instantiate OpenHelper

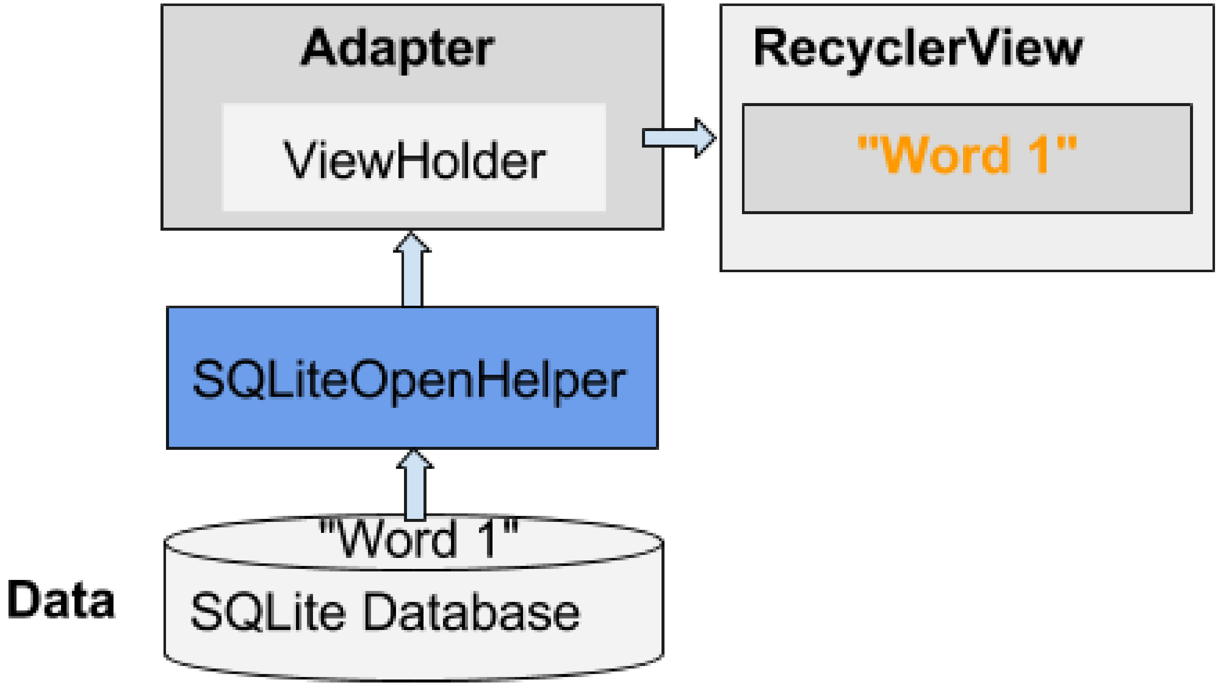
Create an instance of your OpenHelper

- In MainActivity onCreate()

```
mDB = new WordListOpenHelper(this);
```

Working with the Database

Architecture with SQLite database



When to use transactions

- Use transactions when performing multiple operations that all need to complete to keep database consistent
- Use to batch multiple independent operations to improve performance
- Can be nested

Transaction idiom

```
db.beginTransaction();
try {
  ...
  db.setTransactionSuccessful();
} finally {
  db.endTransaction();
}
```

Cloud Backup

- It is a good idea to back up your app's database
- Consider the Cloud Backup options

Contacts App example

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

Step1: Creating Contact 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;}
```

```
         // 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 1: Creating the class

Step2: 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:

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

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

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

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

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()

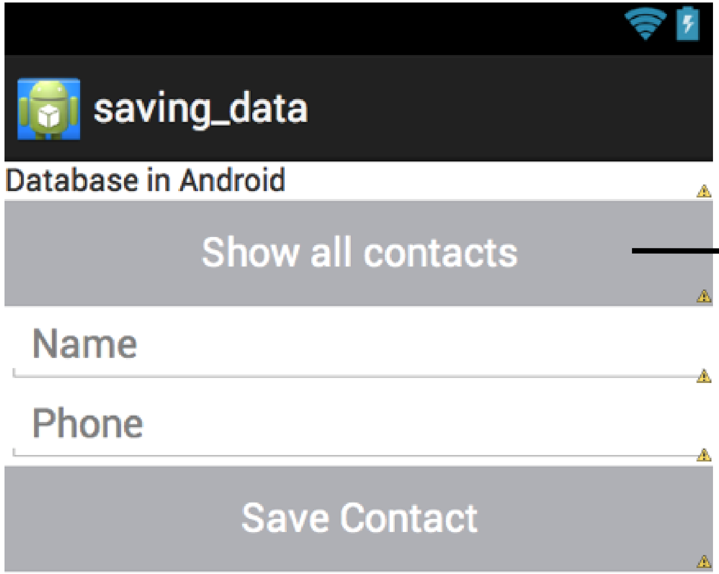
Mandatory metnods	onCreate()
	onUpgrade()

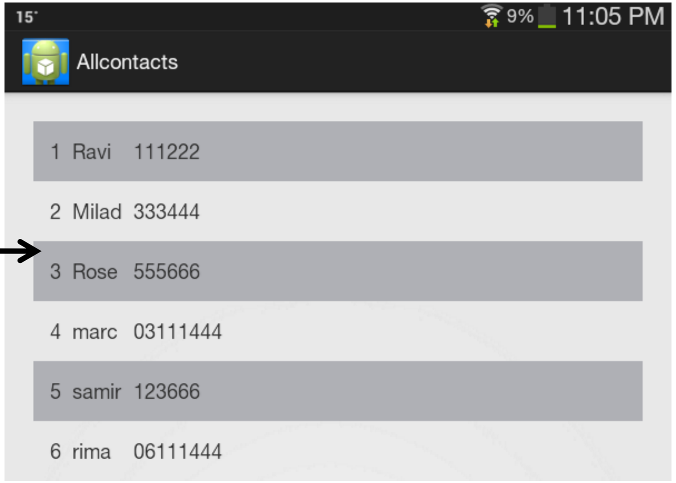
User defined CRUD methods	Addcontact, delete, Getall,… All use the insert, Query, rawQuery, Execsql…

Activity

?

Let’s do it





1	Ravi	111222
2	Milad	333444
3	Rose	555666
4	marc	03111444
5	samir	123666
6	rima	06111444

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 source 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>
```

Source Code for Allcontacts Activity

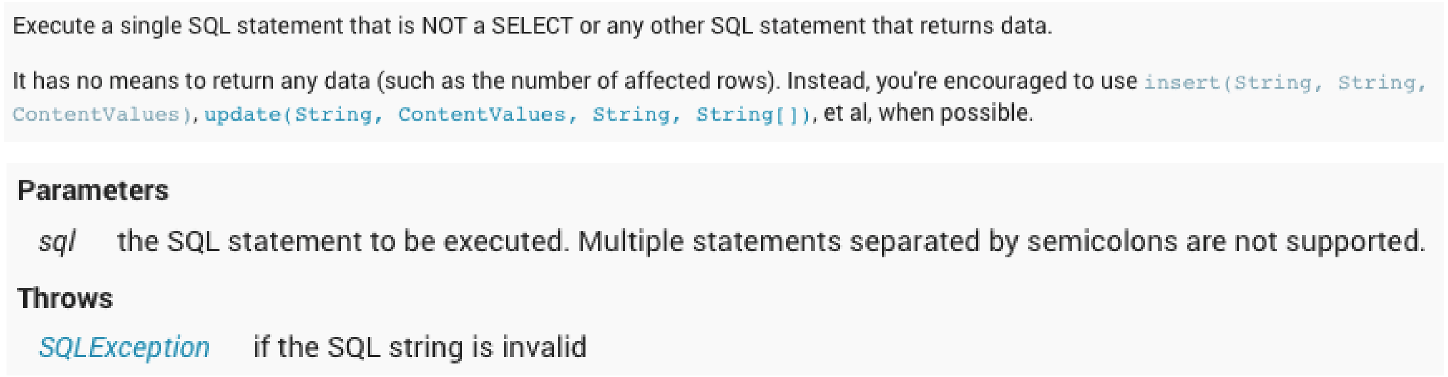
```
public class Allcontacts extends Activity {
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_allcontacts);

                // 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!

END

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

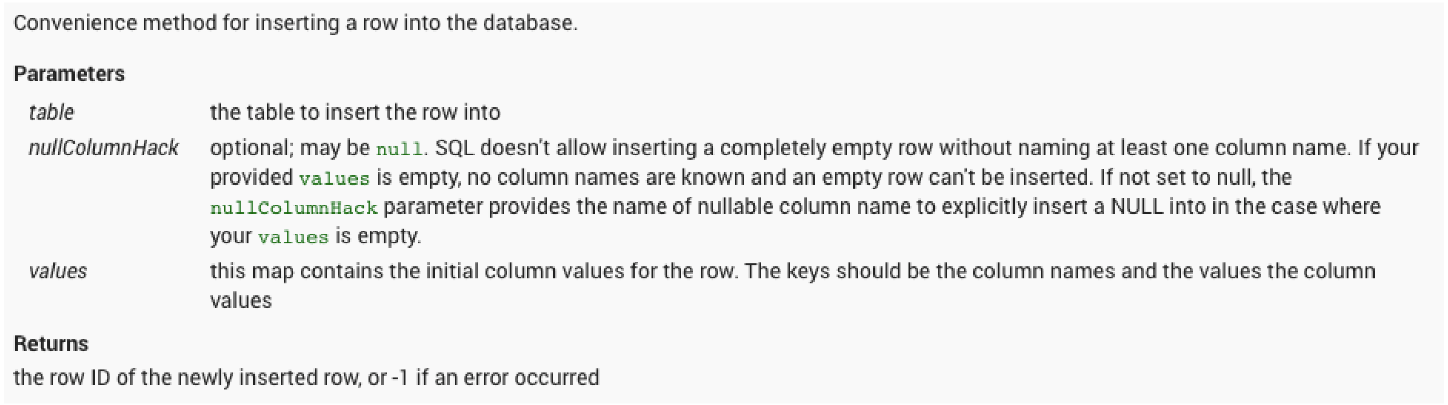
sql	the SQL statement to be executed. Multiple statements separated by semicolons are not supported.

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

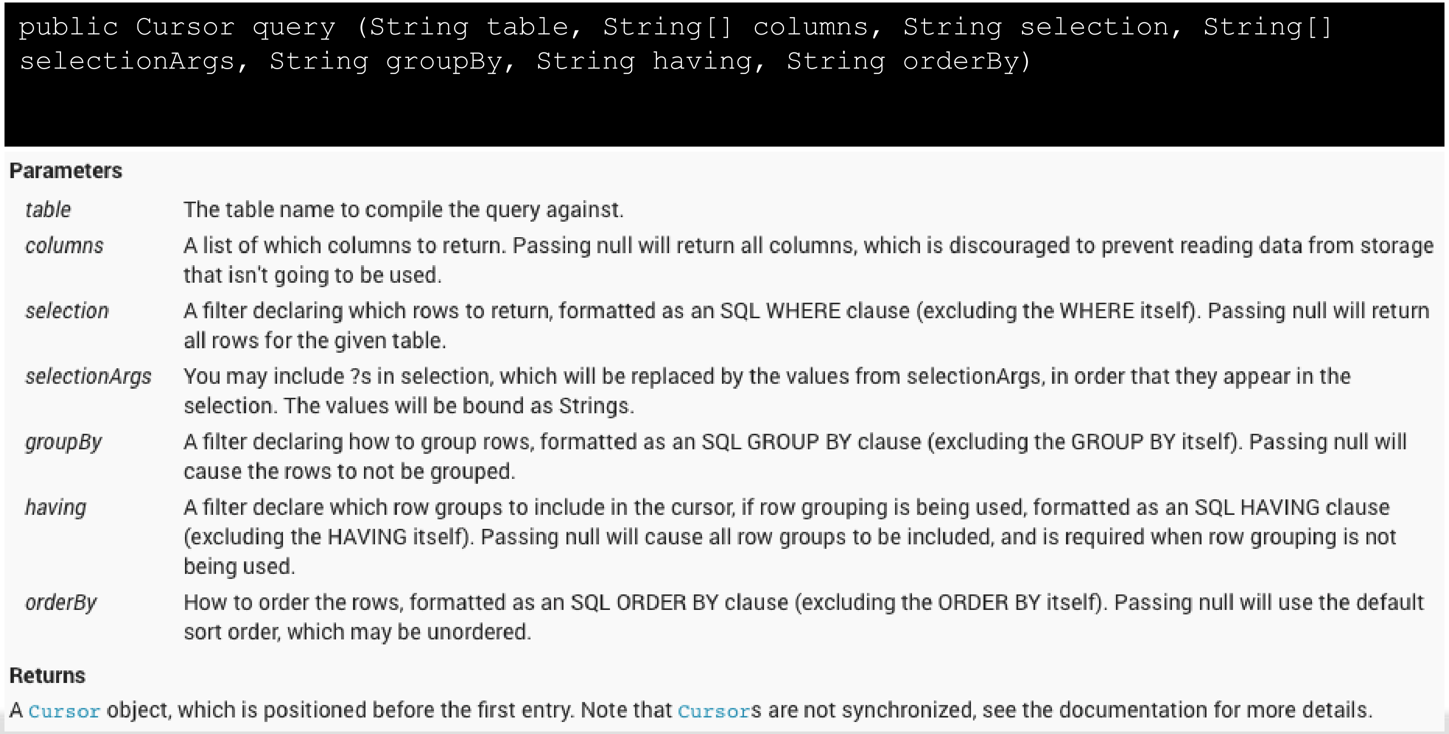
table	the table to insert the row into
nullColumnHack	optional; may be `null`. SQL doesn't allow inserting a completely empty row without naming at least one column name. If your provided `values` is empty, no column names are known and an empty row can't be inserted. If not set to null, the `nullColumnHack` 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

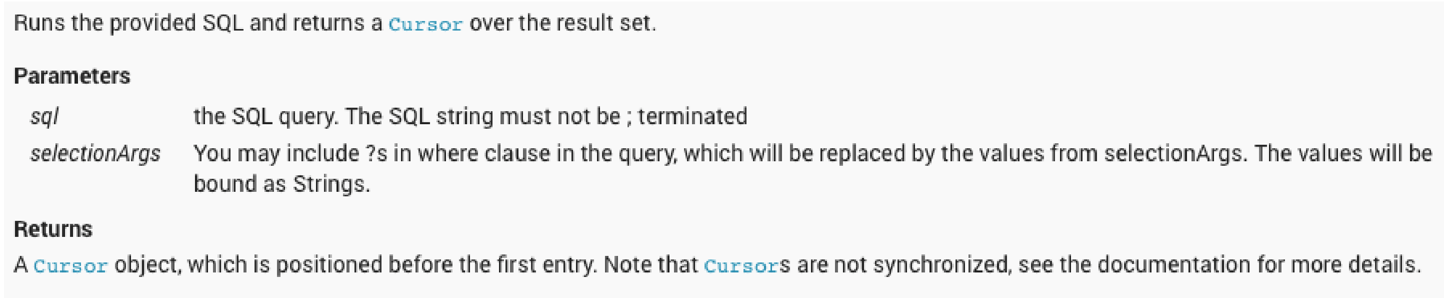
table	The table name to compile the query against.
columns	A list of which columns to return. Passing null will return all columns, which is discouraged to prevent reading data from storage that isn't going to be used.
selection	A filter declaring which rows to return, formatted as an SQL WHERE clause (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

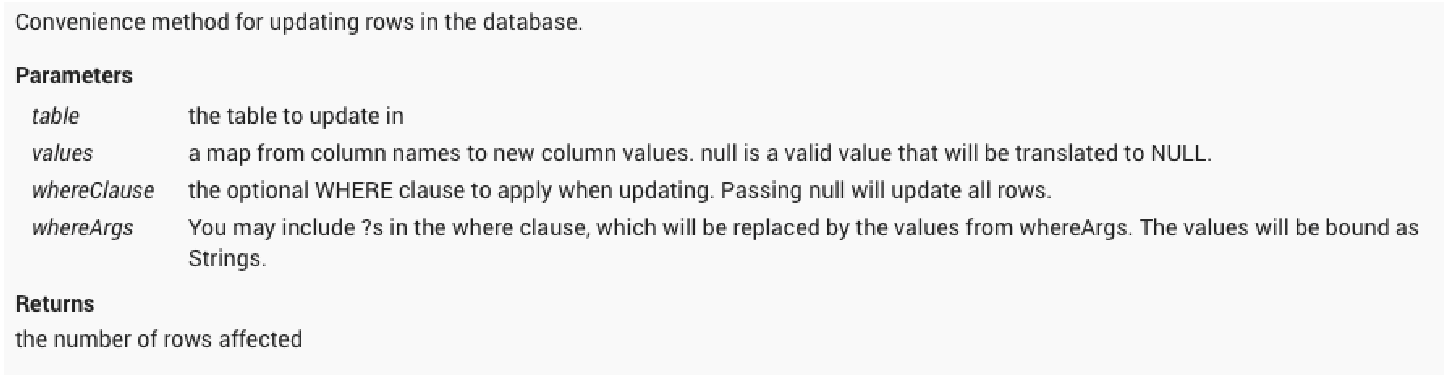
sql	the SQL query. The SQL string must not be ; terminated
selectionArgs	You may include ?s in where clause in the query, which will be replaced by the values from selectionArgs. The values will be bound as Strings.

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