

Lecture – Part B

Persistent Data in Android

DT228/3

Dr. Susan McKeever



Choosing data storage for your app

- Files
- Local SQLite Database
- Remote storage

Pros and cons for each

Complexity of data that can be stored/Ability to query data/
Structure of data/ Storage capacity/Reqmt for internet connection/ Security of data/
Frequency of data changes/ etc

Example: Choosing data storage for your app

A mobile app to allow surveyors to log water table levels in soil; The app will include local maps showing the table levels in the area

- What is your recommendations re persistent data Storage?
- Drawn the technical architecture – including all h/w and s/w components

Example: Choosing data storage for your app -

- Remote storage.. What are the options?
 - Anything that sits on a server that stores data...
 - Formal database e.g. Oracle/ MySQL
 - Google Fusion
 - etc

Doing Database operations

Prone to run time errors

e.g. `execSQL()`.. The SQL statement passed in might be invalid

`getWritableDatabase()` – the system may have a problem connecting to the database etc

Such methods “throw” exceptions – as seen in the API

Error Handling

- Some SQL methods throw exceptions which your code can catch (and if you don't...?)

SQLiteDatabase | Android Developers - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html#execSQL(java.lang.String)

Most Visited Getting Started Latest Headlines Connecting to an Ora...

mary wingfield - Google Search Gmail - Inbox (48) - suemckeever@gma... W Print screen - Wikipedia, the free encyc... SQLiteDatabase | Android Devel...

English Android.com

search developer docs Search

Filter by API Level: 11

Home SDK Dev Guide **Reference** Resources Videos Blog

End a transaction. See beginTransaction for notes about how to use this and when transactions are committed and rolled back.

public void **execSQL** ([String](#) sql) Since: API Level 1

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.

When using [enableWriteAheadLogging\(\)](#), journal_mode is automatically managed by this class. So, do not set journal_mode using "PRAGMA journal_mode" statement if your app is using [enableWriteAheadLogging\(\)](#)

Parameters

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

Throws

[SQLException](#) if the SQL string is invalid

public void **execSQL** ([String](#) sql, [Object\[\]](#) bindArgs) Since: API Level 1

Execute a single SQL statement that is NOT a SELECT/INSERT/UPDATE/DELETE.

Error Handling

- Some SQL methods throw exceptions which your code can catch
- Put this code into try/catch unless your code is robust enough to never hit the error

```
try
{
    // Code that
    execSQL.....
}
catch (SQLException e)
{
    // whatever you want to happen e.g.
    Log.e("Error executing SQL...",
        e.toString());
}
finally
{ ... }
```

Error Handling

- The RETURN values from SQL methods give useful info too...

The screenshot shows the Android Developers website in a Mozilla Firefox browser. The page is titled "SQLiteDatabase | Android Developers" and displays the API documentation for the `insert` method. The method signature is `public long insert (String table, String nullColumnHack, ContentValues values)`, with a note "Since: API Level 1". The description states it is a "Convenience method for inserting a row into the database." The parameters are listed as `table` (the table to insert the row into), `nullColumnHack` (optional, may be `null`), and `values` (this map contains the initial column values for the row). The return value is described as "the row ID of the newly inserted row, or -1 if an error occurred". Below this, the `insertOrThrow` method is also visible, with a similar signature and description. An orange arrow points to the `insertOrThrow` method. The left sidebar shows a tree view of the Android API, with `SQLiteDatabase` selected. The bottom status bar shows "Done".

Opening/connecting to the database -

- We used `getWritableDatabase()` from the `SQLiteOpenHelper` class
- API definition:

public synchronized SQLiteDatabase getWritableDatabase ()

Create and/or open a database that will be used for reading and writing. The first time this is called, the database will be opened and `onCreate(SQLiteDatabase)`, `onUpgrade(SQLiteDatabase, int, int)` and/or `onOpen(SQLiteDatabase)` will be called.

Once opened successfully, the database is cached, so you can call this method every time you need to write to the database. (Make sure to call `close()` when you no longer need the database.) Errors such as bad permissions or a full disk may cause this method to fail, but future attempts may succeed if the problem is fixed.

Database upgrade may take a long time, **you should not call this method from the application main thread**, including from `ContentProvider.onCreate()`.

Returns

- a read/write database object valid until `close()` is called

Throws

SQLiteException if the database cannot be opened for writing

Opening/connecting to the database

- `getWritableDatabase()` to connect to the database
- Can take a long time..
 - E.g. if Database upgrade is involved
- For slow operations (such as this) consider use of threads.
- Subclass a class called **AsyncTask**
 - Asynchronous (background).. Versus synchronous (wait)
 - (like comparing email versus phone call..)..
 - Must override the **doInBackground()** method– which runs on a separate thread
 - Put the “slow” commands in to your own `AsyncTask`

Opening/connecting to the database - Asynchronous Task

■ ■

```
private class MyLongTask extends AsyncTask
{
    @Override
    protected Object doInBackground(final Object... objects)
    {
        try
        {
            openHelperRef.getWritableDatabase();

        }
        catch (etc
```

- Put your slow tasks in it's own asyncTask class – (subclassed)
- Use the **doInBackground()** method to execute the code.. (above)
- Then... instantiate this class from your activity that needs it..
- `new MyLongTask().execute();`

Remember your SQL...

**SELECT
INSERT
UPDATE
DELETE**

SQL for selects

Getting Data from a Database

The Select statement:

```
SELECT column_1 [ , column_N ]  
FROM table_name  
[ WHERE column_1 = 'search term' ];
```

Example:

```
SELECT * FROM tasks;
```

SQL for Updates

Saving Data to a Database

The Update statement:

```
UPDATE table  
SET column_1 = 'new value'  
[ , column_N = 1 ]  
[ WHERE id = 1 ];
```

Example:

```
UPDATE tasks  
SET complete = 'true'  
WHERE id = 1;
```

SQL for inserts

Adding Data to a Database

The Insert statement:

```
INSERT INTO table (column_1 [ , column_N ])  
VALUES (value_1 [ , value_2 ]);
```

Example:

```
INSERT INTO tasks (name, complete)  
VALUES ( 'Get Milk', 'false' );
```

SQL for deletes

Removing Data from a Database

The Delete statement:

```
DELETE FROM table  
[ WHERE id = 1 ];
```

Example:

```
DELETE FROM tasks  
WHERE id = 1;
```


QUERIES – How to call the SELECT

rawQuery()

Simplest

Pass in the SQL SELECT statement directly

Ok for simple queries (e.g. Always have same columns returned)

```
myCursor = db.rawQuery("SELECT _ID, name, city " +  
                        "FROM people WHERE name = " +  
                        personName, null)
```

(Or better – just put your SQL statement into a string variable and pass that in: db.rawQuery(varSql, null)

QUERIES – How to call the **SELECT** query()

Query method builds up the SQL statement from a bunch of parameters

More flexible than `rawQuery()` ...
See overleaf

SQLiteQueryBuilder

Provides a way to “build” queries – see the API

Look at these in the API if `rawQuery()` doesn't meet what you need...

Query() example

query()

Query method builds up the SQL statement from a bunch of parameters e.g.

```
public Cursor getAllPeople()  
{  
    return db.query(DATABASE_TABLE, new String[] {  
        KEY_ROWID,  
        KEY_FIRSTNAME,  
        KEY_SURNAME,  
        KEY_CITY},  
        null,  
        null,  
        null,  
        null,  
        null);  
}
```

Cursors

Query results

- Whatever rows are *returned* comes back in a **Cursor**
- To view the row(s), need to navigate around the cursor.
- API supplies various methods for moving along the rows of the cursor
e.g.

`moveToFirst()`

`moveToNext()`

`isAfterLast()`

Cursor - methods detail from the API

Public boolean `moveToFirst()`

Move the cursor to the first row.

This method will return false if the cursor is empty.

Returns

whether the move succeeded.

Etc

Cursor - iterating through rows..

See Code snippet..

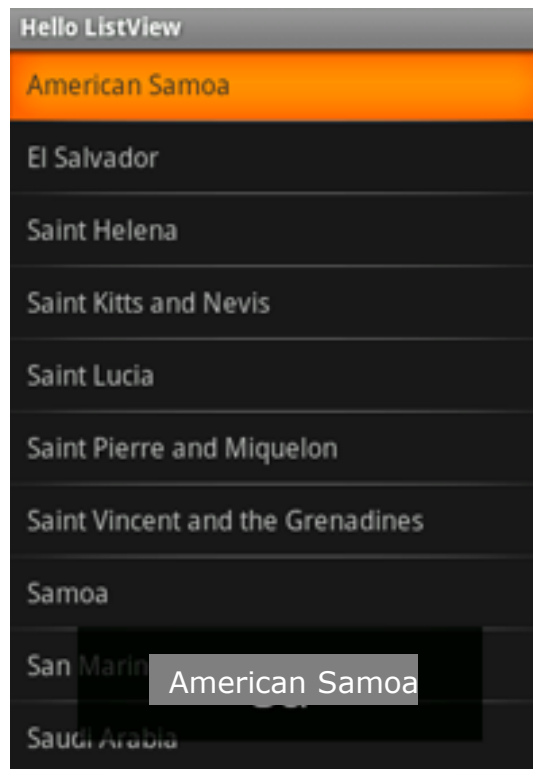
- Return the cursor from the query....
- **IF you want to do something individually with the rows ... need to iterate through the rows in the cursor**
 - Go to the first record
 - Move to the next...
 - Till you hit the last one..
 - Same logic for Android.. Web... Java... etc
- Extract values out of the cursor using getString(index), getInt(index) etc. Where index is the column position in the cursor

Cursor - Displaying database rows on a list

Usually.. Want to display back query results into a list....

**<ListView>
for XML
Layout**

**Row.XML
(maybe)**



SimpleCursorAdaptor

**An
Adapter
for
supplying
the list**

A **cursor of
database
rows for
feeding
into the
adaptor**

SimpleCursorAdapter -

See In Class example

- An off the shelf Adapter from the Android API (such as SimpleCursorAdapter means you don't have to navigate the individual result (cursor) rows..
- Just use the cursor object as a parameter when instantiating the adapter you're going to use

SimpleCursorAdaptor -

VERY similar to previous ArrayAdapter example – Except..it needs:

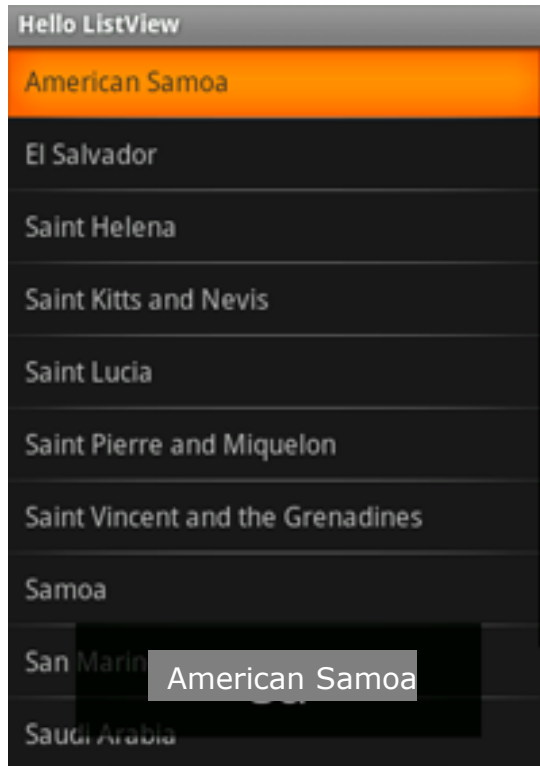
1. a cursor passed in as database source (instead of array)
2. The columns to display from the cursor
3. The textviews to display (the columns)in the layout

```
// the desired columns to be bound
(2) String[] columns = new String[] {"surname", "city"};

// the XML defined views which the data will be bound to
(3) int[] rowIDs = new int [] {R.id.surname_entry,
R.id.city_entry};

// create the adapter using the cursor pointing to the desired
data as well as the layout information
SimpleCursorAdapter mAdapter = new SimpleCursorAdapter(this,
R.layout.row, myCursor, columns, to);
(1)      (2)      (3)
```

Displaying a list from a dB



A cursor adaptor to take the data from a query to a list.. DONE

Then..

How to “do something” when an on the list is clicked? Where does code go?

Responding to list clicks (when populated from a dB)

OnListItemClicked() method as before

- Automatically called in a ListActivity when an item is clicked..
- Usually want to find the contents of the row that was clicked
- If the data your want is in the cursor – use the position parameters to retrieve it.

e.g.

```
public void onListItemClick(ListView parent, View v, int position,
                           long id)
{
    super.onListItemClick(l, v, position, id); // always 1st line
    cursor = (Cursor) mAdapter.getItem(position);
    // retrieve the 2nd column which contains a name..
    String myName = cursor.getString(1);
    // add code to do whatever I want when an item is clicked..etc
```

Putting it all together

e.g.

- Data entry screen to “add” something
- List of “things”
- Click on a list to display details of a thing

What classes do you need?

What you've learnt about dBs

How to use the SQLiteOpenHelper class

How to use inserts, updates, deletes

ContentValues()

Error handling

Asynchronous tasks for opening/connecting

How to do queries – and use cursors..

How to use populate a list using

SimpleCursorAdaptor

How to get at the list row clicked..

How to view a SQLite database