



Persistence: Files & Preferences SQL Databases

ONE LOVE. ONE FUTURE.

Android Files

Persistence is a strategy that allows the reusing of volatile objects and other data items by storing them into a permanent storage system such as disk files and databases.

File IO management in Android includes –among others- the familiar IO Java classes: Streams, Scanner, PrintWriter, and so on.

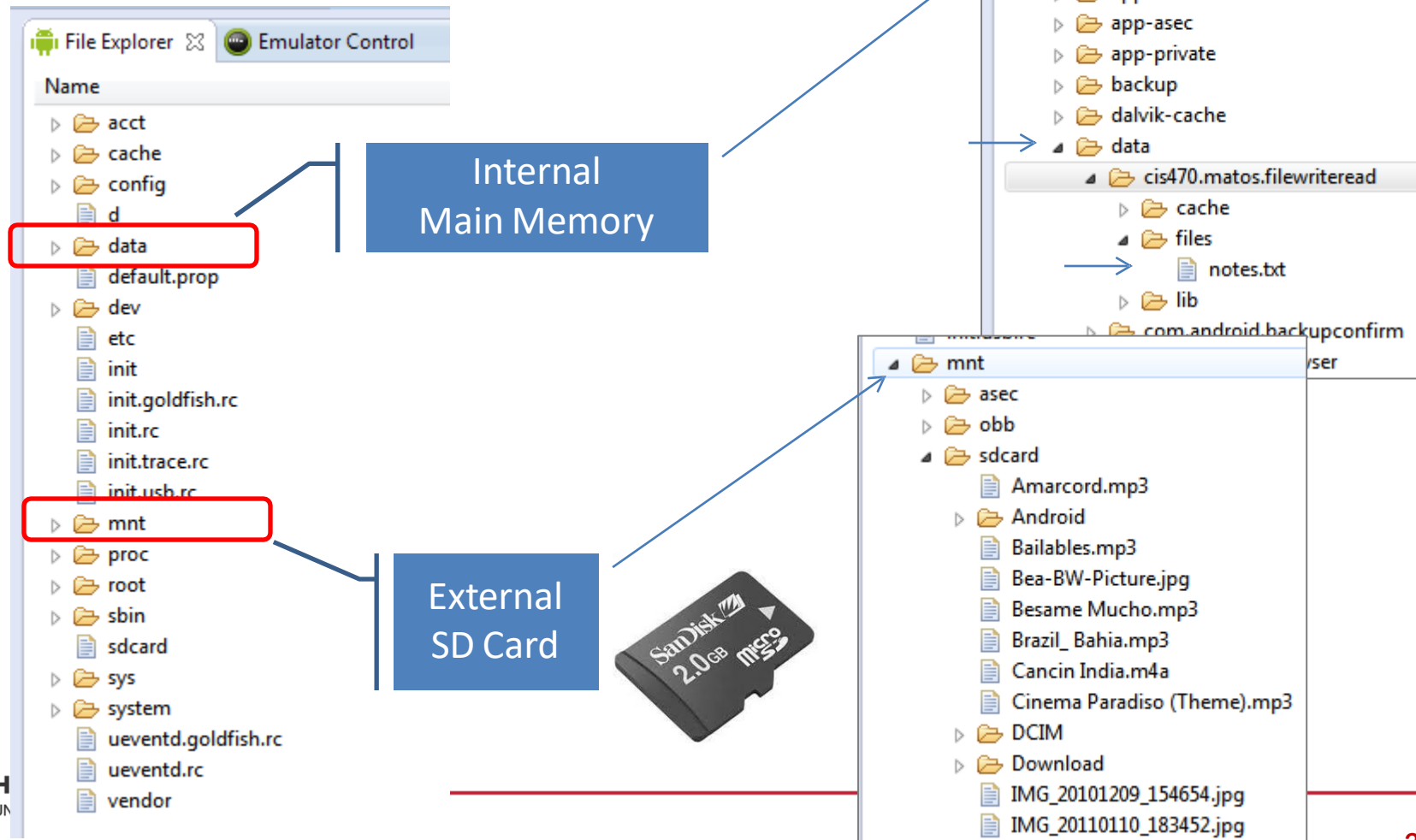
Permanent files can be stored *internally* in the device's main memory (usually small, but not volatile) or *externally* in the much larger SD card.

Files stored in the device's memory, share space with other application's resources such as code, icons, pictures, music, etc.

Internal files are called: **Resource Files** or **Embedded Files**.

Exploring Android's File System

Use the emulator's **File Explorer** to see and manage your device's storage structure.



Choosing a Persistent Environment

Your permanent data storage destination is usually determined by parameters such as:

- size (**small**/**large**),
- location (**internal**/**external**),
- accessibility (**private**/**public**).

Depending of your situation the following options are available:

1. Shared Preferences

Store private primitive data in *key-value* pairs.

2. Internal Storage

Store private data on the device's main memory.

3. External Storage

Store public data on the shared external storage.

4. SQLite Databases

Store structured data in a private/public database.

5. Network Connection

Store data on the web.

Shared Preferences

SharedPreferences files are good for handling a handful of Items. Data in this type of container is saved as **<Key, Value>** pairs where the *key* is a string and its associated *value* must be a primitive data type.

KEY	VALUE

This class is functionally similar to Java Maps, however; unlike Maps they are *permanent*.

Data is stored in the device's internal main memory.

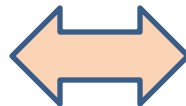
PREFERENCES are typically used to keep state information and shared data among several activities of an application.

Shared Preferences

- Using Preferences API calls
- Each of the Preference mutator methods carries a typed-value content that can be manipulated by an editor that allows putXxx... and getXxx... commands to place data in and out of the Preference container.

- Xxx = { Long, Int, Double, Boolean, String }

Preference Container	
Key	Value



.getXxx(key_n)
.getAll()
.getStringSet()
...

.putXxx(key_n, value_n)

.remove(key_n)
.clear()
.commit()
.apply()

Example: Shared Preferences

In this example the user selects a preferred 'color' and 'number'. Both values are stored in a SharedPreferences file.

```
fun usingPreferences() {  
    // Save data in a SharedPreferences container  
    // We need an Editor object to make preference changes.  
    ① → val myPrefs = getSharedPreferences("my_preferred_choices", MODE_PRIVATE)  
    val editor: SharedPreferences.Editor = myPrefs.edit()  
    ② → editor.putString("chosenColor", "RED")  
    editor.putInt("chosenNumber", 7)  
    editor.apply()  
  
    // retrieving data from SharedPreferences container (apply default if needed)  
    ③ → val favoriteColor = myPrefs.getString("chosenColor", "BLACK");  
    val favoriteNumber = myPrefs.getInt("chosenNumber", 11)  
}
```

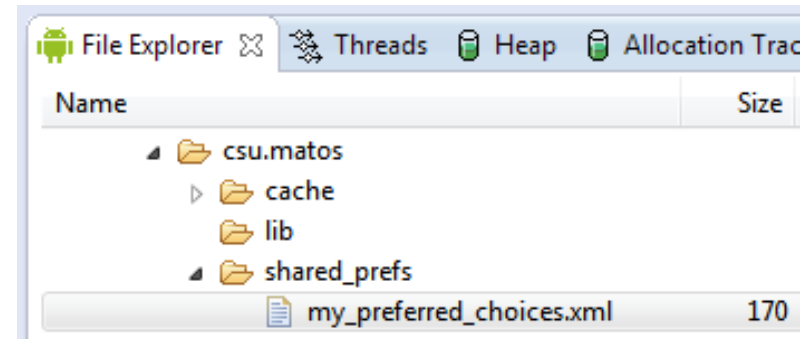
Shared Preferences. Example - Comments

1. The method `getSharedPreferences(...)` creates (or retrieves) a table called *my_preferred_choices* file, using the default **MODE_PRIVATE** access. Under this access mode only the calling application can operate on the file.
2. A `SharedPreferences` editor is needed to make any changes on the file. For instance `editor.putString("chosenColor", "RED")` creates (or updates) the key "chosenColor" and assigns to it the value "RED". All editing actions must be explicitly committed for the file to be updated.
3. The method **getXXX(...)** is used to extract a value for a given key. If no key exists for the supplied name, the method uses the designated default value. For instance `myPrefs.getString("chosenColor", "BLACK")` looks into the file *myPrefs* for the key "chosenColor" to returns its value, however if the key is not found it returns the default value "BLACK".

Shared Preferences. Example - Comments

SharedPreferences containers are saved as XML files in the application's internal memory space. The path to a preference files is
`/data/data/packageName/shared_prefs/filename.`

For instance in this example we have:



If you pull the file from the device, you will see the following

```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
- <map>
    <string name="favorite_color">#ff0000ff</string>
    <int name="favorite_number" value="101"/>
</map>
```

Files & Preferences

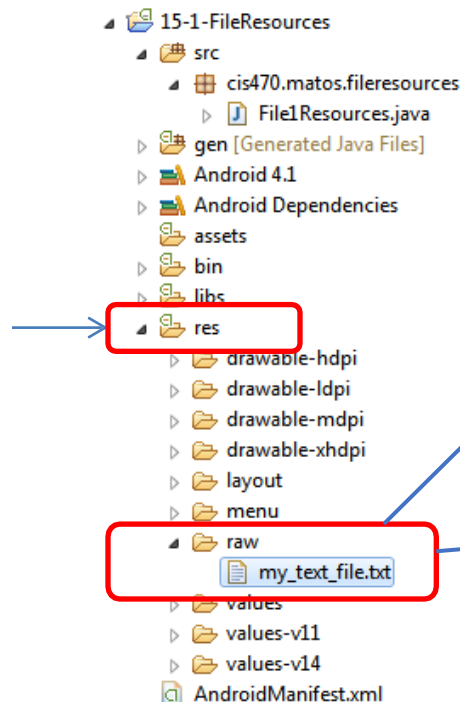
Internal Storage. Reading an Internal Resource File

An Android application may include resource elements such as those in:

res/drawable , **res/raw**, **res/menu**, **res/style**, etc.

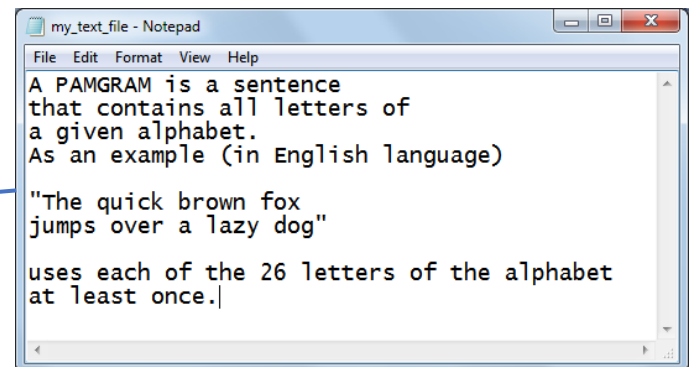
Resources could be accessed through the **.getResources(...)** method. The method's argument is the ID assigned by Android to the element in the R resource file. For example:

```
val inputStream: InputStream = resources.openRawResource(R.raw.my_text_file)
```



If needed create the **res/raw** folder.

Use drag/drop to place the file **my_text_file.txt** in **res** folder. It will be stored in the device's memory as part of the .apk



Example: Reading an Internal Resource File

This app stores a text file in its RESOURCE (**res/raw**) folder.
The embedded raw data is read and displayed in a text box.

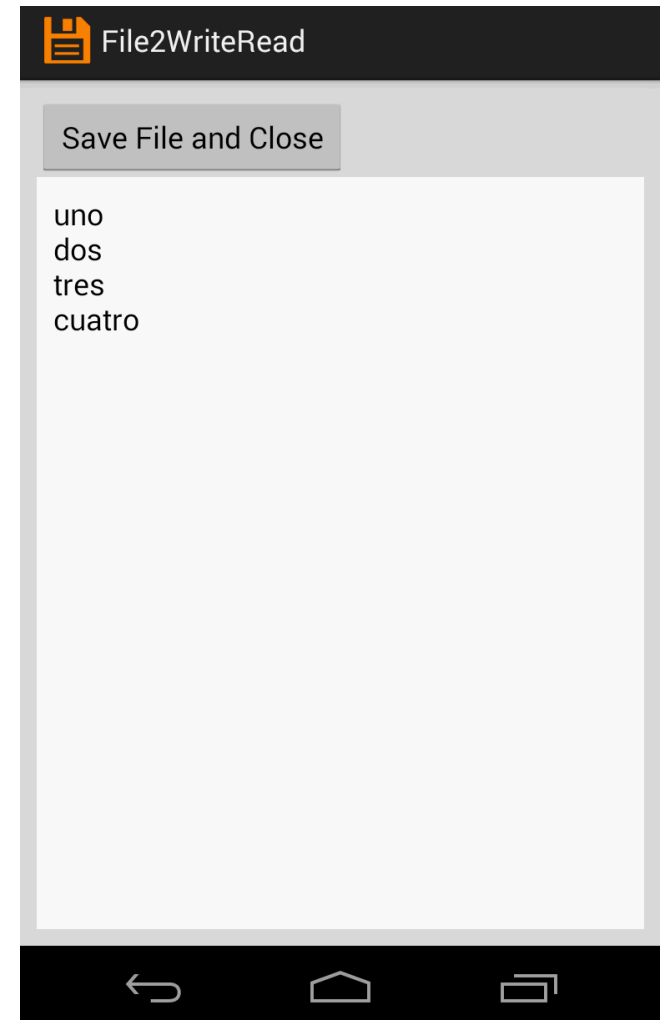
```
val inputStream: InputStream = resources.openRawResource(R.raw.my_text_file)
val reader: InputStreamReader = inputStream.reader()
val content = reader.readText()
reader.close()

binding.textRaw.text = content
```

Example: Reading / Writing an Internal File

In this example an application exposes a GUI on which the user enters a few lines of data. The app collects the input lines and **writes** them to a persistent **internal data file**.

Next time the application is executed, the *internal file* will be **read** and its data will be shown on the UI.



Writing to an internal file

```
val outputStream: OutputStream = openFileOutput("data.txt", MODE_PRIVATE)
// Output file will be saved to /data/data/vn.edu.hust.fileexamples/files/data.txt
val writer: OutputStreamWriter = outputStream.writer()
writer.write(binding.editText.text.toString())
writer.close()
```

Reading from an internal file

```
val inputStream: InputStream = openFileInput("data.txt")
val reader: InputStreamReader = inputStream.reader()
val content: String = reader.readText()
reader.close()

binding.textRaw.text = content
```

Delete an internal file

```
val file: File = File(filesDir.path + "/data.txt")
if (file.exists()) {
    if (file.delete())
        Toast.makeText(this, "File is deleted successfully.",
Toast.LENGTH_LONG).show()
    else
        Toast.makeText(this, "File is deleted failed.", Toast.LENGTH_LONG).show()
} else
    Toast.makeText(this, "File is not existed.", Toast.LENGTH_LONG).show()
```

Copy binary file from resource to internal storage

```
val inputStream = resources.openRawResource(R.raw.test)
val outputStream = openFileOutput("test.txt", MODE_PRIVATE)
val buffer = ByteArray(2048)
while (true) {
    val len = inputStream.read(buffer, 0, buffer.size)
    if (len <= 0)
        break
    outputStream.write(buffer, 0, len)
}
inputStream.close()
outputStream.close()
```

Reading / Writing External SD Files

SD cards offer the advantage of a *much larger capacity* as well as *portability*.

Many devices allow SD cards to be easily removed and reused in another device.

SD cards are ideal for keeping your collection of music, picture, ebooks, and video files.



Reading / Writing External SD Files

Use the **Device Explorer** tool to locate files in your physical device (or emulator). External storage is managed differently depend on Android OS version.

=> **HARD CHALLENGE!!!**

2 types of sd card:

- **Emulated**: available for all type of smartphone models, mounted to /sdcard for compatibility
- **Physical**: almost available on old models

Pixel 2 API 34 Android API 34		
Files Processes		
Name	Permissions	Date
▼ storage	drwx--x---	2023-12-05 14:38
▼ 0F0B-3C05	drwxrwx---	1970-01-01 07:00
> Alarms	drwxrwx---	2023-12-05 14:38
> Android	drwxrwx---	2023-12-05 14:38
> Audiobooks	drwxrwx---	2023-12-05 14:38
> DCIM	drwxrwx---	2023-12-05 14:38
> Documents	drwxrwx---	2023-12-05 14:38
> Download	drwxrwx---	2023-12-05 14:38
> LOST.DIR	drwxrwx---	2023-12-05 14:38
> Movies	drwxrwx---	2023-12-05 14:38
> Music	drwxrwx---	2023-12-05 14:38
> Notifications	drwxrwx---	2023-12-05 14:38
> Pictures	drwxrwx---	2023-12-05 14:38
> Podcasts	drwxrwx---	2023-12-05 14:38
> Recordings	drwxrwx---	2023-12-05 14:38
> Ringtones	drwxrwx---	2023-12-05 14:38
▼ emulated	drwxrwx---	2023-12-05 14:37
▼ 0	drwxrws---	2023-12-05 14:38
> Alarms	drwxrws---	2023-12-05 14:38
> Android	drwxrws--x	2023-12-05 14:37
> Audiobooks	drwxrws---	2023-12-05 14:38
> DCIM	drwxrws---	2023-12-05 14:38
> Documents	drwxrws---	2023-12-05 14:38
> Download	drwxrws---	2023-12-05 14:54
> Movies	drwxrws---	2023-12-05 14:38
> Music	drwxrws---	2023-12-05 14:38

Reading /Writing External SD Files

Although you may use the specific path to an SD file, such as:

`/sdcard/myfile.txt`

it is a better practice to determine the SD location as suggested below

```
val sdPath = Environment.getExternalStorageDirectory().path
```

WARNING

When you deal with external files you need to request permission to read and write. Add the following clauses to your AndroidManifest.xml (**apply for API level below 33**)

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Reading / Writing External SD Files

From Android 6.0 (API Level 23), app needs to ask permission from user.

```
if (Build.VERSION.SDK_INT > Build.VERSION_CODES.M) {  
    if (checkSelfPermission(Manifest.permission.WRITE_EXTERNAL_STORAGE) !=  
        PackageManager.PERMISSION_GRANTED) {  
        requestPermissions(arrayOf(Manifest.permission.WRITE_EXTERNAL_STORAGE), 1234)  
    }  
}
```

```
override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out  
String>, grantResults: IntArray) {  
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)  
    if (requestCode == 1234) {  
        if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {  
            Log.v("TAG", "Permission granted!")  
        } else  
            Log.v("TAG", "Permission denied!")  
    }  
}
```

In Android 10 (API Level 29), use “requestLegacyExternalStorage” to read and write external storage.

Reading / Writing External SD Files

From Android 11 (API Level 30), an app can request All files access from the user by doing the following:

1. Declare the **MANAGE_EXTERNAL_STORAGE** permission in the manifest.
2. Use the **ACTION_MANAGE_ALL_FILES_ACCESS_PERMISSION** intent action to direct users to a system settings page where they can enable the following option for your app: Allow access to manage all files.

To determine whether your app has been granted the **MANAGE_EXTERNAL_STORAGE** permission, call `Environment.isExternalStorageManager()`.

<https://developer.android.com/training/data-storage/manage-all-files>

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {  
    if (!Environment.isExternalStorageManager()) {  
        val intent = Intent(Settings.ACTION_MANAGE_ALL_FILES_ACCESS_PERMISSION)  
        startActivity(intent)  
    }  
}
```

Writing to an external file

```
val path = Environment.getExternalStorageDirectory().path
val file = File("${path}/data.txt")
val outputStream: OutputStream = file.outputStream()
val writer: OutputStreamWriter = outputStream.writer()
writer.write(binding.editText.text.toString())
writer.close()
```

Reading from an external file

```
val path = Environment.getExternalStorageDirectory().path
val file = File("${path}/data.txt")
val inputStream: InputStream = file.inputStream()
val reader: InputStreamReader = inputStream.reader()
val content: String = reader.readText()
reader.close()

binding.editText.setText(content)
```

Copy from internal storage to the shared storage

```
val fromPath = filesDir.path + "/data.txt"
val toPath = Environment.getExternalStorageDirectory().path + "/data.txt"
val fromFile = File(fromPath)
val toFile = File(toPath)
fromFile.copyTo(toFile)
```

List all file and folder in the root directory of the shared storage

```
val root = Environment.getExternalStorageDirectory()
for (entry in root.listFiles()) {
    if (entry.isDirectory)
        Log.v("TAG", "Directory")
    else if (entry.isFile)
        Log.v("TAG", "File")
    Log.v("TAG", entry.path)
}
```

Using SQL databases in Andorid

Included into the core Android architecture there is an standalone Database Management System (DBMS) called **SQLite** which can be used to:

Create a database,

Define

SQL tables,

indices,

queries,

views,

triggers

Insert rows,

Delete rows,

Change rows,

Run queries and

Administer a SQLite database file.



Characteristics of SQLite

- Transactional SQL database engine.
- Small footprint (less than 400KBytes)
- Serverless
- Zero-configuration
- The source code for SQLite is in the public domain.
- According to their website, SQLite is the *most widely deployed SQL database engine in the world* .

Reference:

<http://sqlite.org/index.html>

Creating a SQLite database - Method 1

```
SQLiteDatabase.openDatabase( myDbPath,  
                             null,  
                             SQLiteDatabase.CREATE_IF_NECESSARY);
```

If the database does not exist then create a new one. Otherwise, open the existing database according to the flags:

OPEN_READWRITE, OPEN_READONLY, CREATE_IF_NECESSARY .

Parameters

path to database file to open and/or create

factory an optional factory class that is called to instantiate a cursor when query is called, or *null* for default

flags to control database access mode

Returns the newly opened database

Throws *SQLException* if the database cannot be opened

Example1: Creating a SQLite database - Method 1

```
class MainActivity : AppCompatActivity() {  
    lateinit var db: SQLiteDatabase  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
        val binding = ActivityMainBinding.inflate(layoutInflater)  
        setContentView(binding.root)  
  
        db = SQLiteDatabase.openDatabase(filesDir.path + "/mydb", null,  
            SQLiteDatabase.CREATE_IF_NECESSARY)  
    }  
  
    override fun onStop() {  
        db.close()  
        super.onStop()  
    }  
}
```

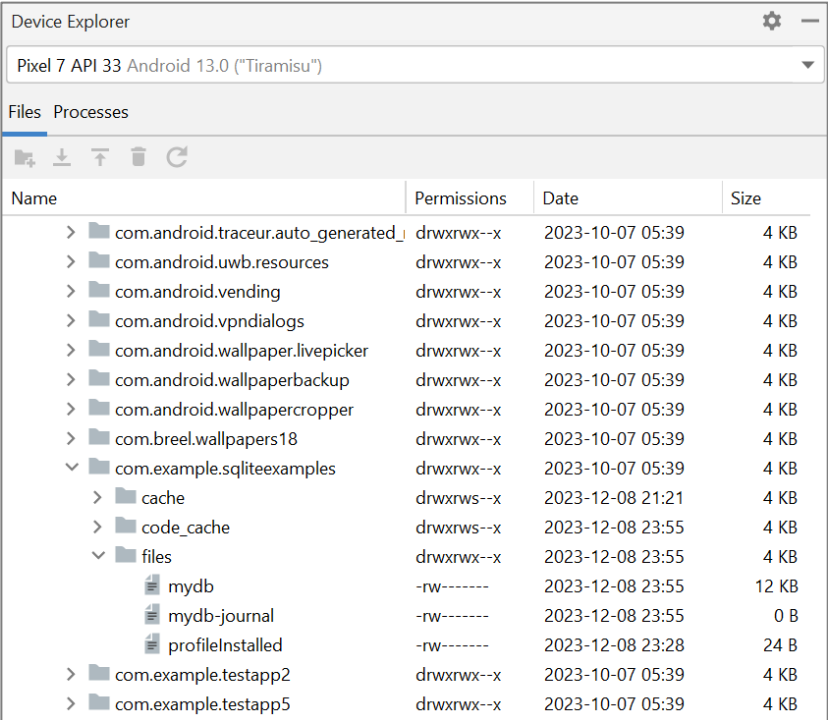
Example1: Creating a SQLite database - Using Memory

SQLite Database is stored using Internal Memory

Path:

`/data/data/com.example.sqliteexamples/files/`

Where: `com.example.sqliteexamples` is the package's name



Name	Permissions	Date	Size
> com.android.traceur.auto_generated	drwxrwx--x	2023-10-07 05:39	4 KB
> com.android.uwb.resources	drwxrwx--x	2023-10-07 05:39	4 KB
> com.android.vending	drwxrwx--x	2023-10-07 05:39	4 KB
> com.android.vpndialogs	drwxrwx--x	2023-10-07 05:39	4 KB
> com.android.wallpaper.livepicker	drwxrwx--x	2023-10-07 05:39	4 KB
> com.android.wallpaperbackup	drwxrwx--x	2023-10-07 05:39	4 KB
> com.android.wallpapercropper	drwxrwx--x	2023-10-07 05:39	4 KB
> com.breel.wallpapers18	drwxrwx--x	2023-10-07 05:39	4 KB
▼ com.example.sqliteexamples	drwxrwx--x	2023-10-07 05:39	4 KB
> cache	drwxrws--x	2023-12-08 21:21	4 KB
> code_cache	drwxrws--x	2023-12-08 23:55	4 KB
▼ files	drwxrwx--x	2023-12-08 23:55	4 KB
mydb	-rw-----	2023-12-08 23:55	12 KB
mydb-journal	-rw-----	2023-12-08 23:55	0 B
profileInstalled	-rw-----	2023-12-08 23:28	24 B
> com.example.testapp2	drwxrwx--x	2023-10-07 05:39	4 KB
> com.example.testapp5	drwxrwx--x	2023-10-07 05:39	4 KB

An Alternative Method: `openOrCreateDatabase`

An alternative way of opening/creating a SQLite database in your local Android's internal data space is given below

```
val db: SQLiteDatabase = openOrCreateDatabase(  
    "myfriendsDB",  
    MODE_PRIVATE,  
    null)
```

Assume app is made in a namespace called `com.example.sqliteexamples`, then the full name of the newly created database file will be:

`/data/data/com.example.sqliteexamples/database/myfriendsDB`

- The file can be accessed by all components of the same application.
- Other **MODE** values: `MODE_WORLD_READABLE`, and `MODE_WORLD_WRITEABLE` were deprecated on API Level 17.
- **null** refers to optional **factory** class parameter (skip for now)

Type of SQL Commands

Once created, the SQLite database is ready for normal operations such as: *creating, altering, dropping resources (tables, indices, triggers, views, queries etc.) or administrating database resources (containers, users, ...).*

Action queries and **Retrieval queries** represent the most common operations against the database.

- A **retrieval query** is typically a *SQL-Select* command in which a table holding a number of fields and rows is produced as an answer to a data request.
- An **action query** usually performs maintenance and administrative tasks such as manipulating tables, users, environment, etc.

Transaction Processing

Transactions are desirable because they help maintaining consistent data and prevent unwanted data losses due to abnormal termination of execution.

In general it is convenient to process **action queries** inside the protective frame of a **database transaction** in which the policy of “*complete success or total failure*” is transparently enforced.

*This notion is called: **atomicity** to reflect that all parts of a method are fused in an indivisible ‘statement’.*

Transaction Processing

The typical Android's way of running transactions on a SQLiteDatabase is illustrated by the following code fragment (Assume **db** is a SQLiteDatabase)

```
db.beginTransaction()  
try {  
    //perform your database operations here ...  
    db.setTransactionSuccessful() //commit your changes  
}  
catch (e: SQLException) {  
    //report problem  
}  
finally {  
    db.endTransaction()  
}
```

The transaction is defined between the methods: *beginTransaction* and *endTransaction*. You need to issue the *setTransactionSuccessful()* call to commit any changes. The absence of it provokes an implicit *rollback* operation; consequently *the database is reset to the state previous to the beginning of the transaction*.

Create and Populate a SQL Table

The **SQL** syntax used for creating and populating a table is illustrated in the following examples

```
create table tblAMIGO(  
    recID integer PRIMARY KEY autoincrement,  
    name text,  
    phone text );
```

```
insert into tblAMIGO(name, phone) values ('AAA', '555-1111' );
```

The *autoincrement* value for *recID* is NOT supplied in the insert statement as it is internally assigned by the DBMS.

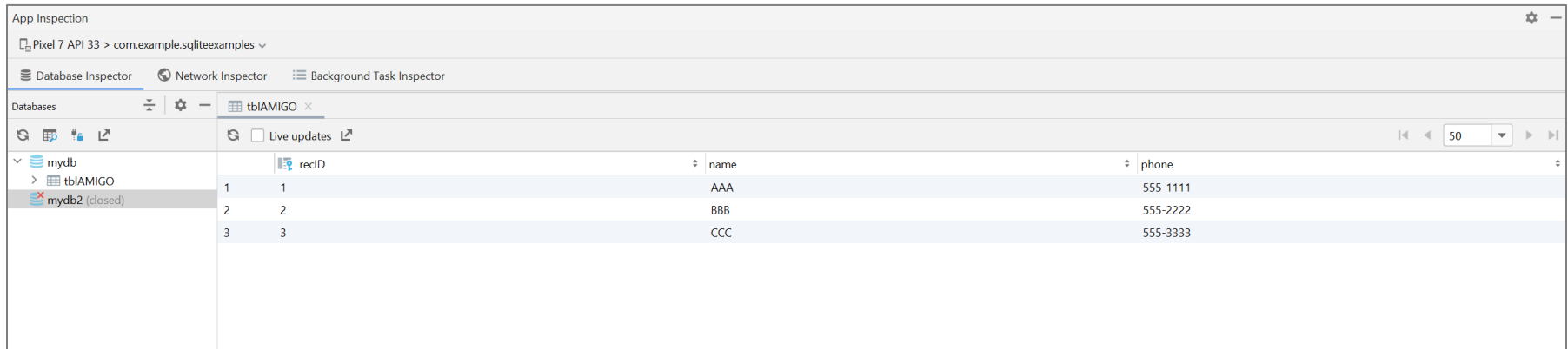
Example 2. Create and Populate a SQL Table

- Our Android app will use the **execSQL(...)** method to manipulate SQL *action queries*. The example below creates a new table called **tblAmigo**.
- The table has three fields: a numeric unique identifier called **recID**, and two string fields representing our friend's **name** and **phone**.
- If a table with such a name exists it is first dropped and then created again.
- Finally three rows are inserted in the table

```
db.execSQL("create table tblAMIGO(" +  
    "recID integer primary key autoincrement," +  
    "name text," +  
    "phone text);")  
db.execSQL("insert into tblAMIGO(name, phone) values('AAA', '555-1111');")  
db.execSQL("insert into tblAMIGO(name, phone) values('BBB', '555-2222');")  
db.execSQL("insert into tblAMIGO(name, phone) values('CCC', '555-3333');")
```

Example 2. Create and Populate a SQL Table

Use **Database Inspector** tool (in **App Inspection**) to monitor the database when app is running.



Example 2. Create and Populate a SQL Table

Comments

1. The field **recID** is defined as the table's **PRIMARY KEY**.
2. The “*autoincrement*” feature guarantees that each new record will be given a unique serial number (0,1,2,...).
3. On par with other SQL systems, SQLite offers the data types: ***text, varchar, integer, float, numeric, date, time, timestamp, blob, boolean.***
3. In general any well-formed DML SQL action command (insert, delete, update, create, drop, alter, etc.) could be framed inside an **execSQL(...)** method call.

Caution:

You should call the **execSQL** method inside of a ***try-catch-finally*** block. Be aware of potential **SQLiteException** conflicts thrown by the method.

Asking Questions - SQL Queries

1. **Retrieval queries** are known as *SQL-select* statements.
2. *Answers* produced by retrieval queries are always held in a *table*.
3. In order to process the resulting table rows, the user should provide a **cursor** device. Cursors allow a *row-at-the-time access* mechanism on SQL tables.

Android-SQLite offers two strategies for phrasing *select* statements: ***rawQueries*** and ***simple queries***. Both return a database *cursor*.

1. **Raw queries** take for input any (syntactically correct) SQL-select statement. The select query could be as complex as needed and involve any number of tables (only a few exceptions such as outer-joins)
2. **Simple queries** are compact *parametized* lookup functions that operate on a single table (for developers who prefer not to use SQL).

SQL Select Statement – Syntax <http://www.sqlite.org/lang.html>

```
select    field1, field2, ... , fieldn
from      table1, table2, ... , tablen
```

```
where     ( restriction-join-conditions )
order by  fieldn1, ..., fieldnm
group by  fieldm1, ... , fieldmk
having     (group-condition)
```

The first two lines are mandatory, the rest is optional.

1. The *select* clause indicates the fields to be included in the answer
2. The *from* clause lists the tables used in obtaining the answer
3. The *where* component states the conditions that records must satisfy in order to be included in the output.
4. *Order by* tells the sorted sequence on which output rows will be presented
5. *Group by* is used to partition the tables and create sub-groups
6. *Having* formulates a condition that sub-groups made by partitioning need to satisfy.

Two Examples of SQL-Select Statements

Example A.

```
SELECT    LastName, cellPhone
FROM      ClientTable
WHERE     state = 'Ohio'
ORDER BY  LastName
```

Example B.

```
SELECT    city, count(*) as TotalClients
FROM      ClientTable
GROUP BY  city
```

Example3. Using a Parameterless RawQuery (version 1)

Consider the following code fragment

```
val c1: Cursor = db.rawQuery("select * from tblAMIGO", null)
```

1. The previous *rawQuery* contains a select-statement that retrieves all the rows (and all the columns) stored in the table `tblAMIGO`. The resulting table is wrapped by a **Cursor** object `c1`.
2. The 'select *' clause instructs SQL to grab all-columns held in a row.
3. Cursor `c1` will be used to traverse the rows of the resulting table.
4. Fetching a row using cursor `c1` requires advancing to the next record in the answer set (cursors are explained a little later in this section).
5. Fields provided by SQL must be bound to local Java variables (soon we will see to that).

Example3. Using a Parametized RawQuery (version 2)

Passing arguments.

Assume we want to count how many friends are there whose name is 'BBB' and their recID > 1. We could use the following solution:

```
val sql = "select count(*) as Total " +  
          "from tblAMIGO " +  
          "where recID > ? and name = ?"  
val args = arrayOf("1", "BBB")  
val cs = db.rawQuery(sql, args)
```

The various symbols '?' in the SQL statement represent positional placeholders. When **.rawQuery()** is called, the system binds each empty placeholder '?' with the supplied **args**-value. Here the first '?' will be replaced by "1" and the second by "BBB".

Example3. Using a Stitched RawQuery (version 3)

As in the previous example, assume we want to count how many friends are there whose name is 'BBB' and their recID > 1. We could use the following solution:

```
val args = arrayOf("1", "BBB")
val sql = "select count(*) as Total " +
    "from tblAMIGO " +
    "where recID > ${args[0]} and name = ${args[1]}"
val cs = db.rawQuery(sql, null)
```

Instead of the symbols '?' acting as placeholder, we conveniently concatenate the necessary data fragments during the assembling of our SQL statement.

SQL Cursors

Cursors are used to gain sequential & random access to tables produced by SQL *select* statements.

Cursors support *one row-at-the-time* operations on a table. Although in some DBMS systems cursors can be used to update the underlying dataset, the SQLite version of cursors is **read-only**.



Cursors include several types of operators, among them:

1. **Positional awareness:** isFirst(), isLast(), isBeforeFirst(), isAfterLast().
2. **Record navigation:** moveToFirst(), moveToLast(), moveToNext(), moveToPrevious(), move(n).
3. **Field extraction:** getInt, getString, getFloat, getBlob, getDouble, etc.
4. **Schema inspection:** getColumnName(), getColumnNames(), getColumnIndex(), getColumnCount(), getCount().

Example 4A. Traversing a Cursor – Simple Case

```
val cs = db.rawQuery("select * from tblAMIGO", null)
cs.moveToFirst()
do {
    val recID = cs.getInt(0)
    val name = cs.getString(1)
    val phoneIndex = cs.getColumnIndex("phone")
    val phone = if (phoneIndex > -1) cs.getString(phoneIndex) else ""
    Log.v("TAG", "$recID - $name - $phone")
} while (cs.moveToNext())
cs.close()
```

1. Prepare a `rawQuery` passing a simple sql statement with no arguments, catch the resulting tuples in cursor `cs`.
2. Move the fetch marker to the absolute position prior to the first row in the file. The valid range of values is $-1 \leq \text{position} \leq \text{count}$.
3. Use **`moveToNext()`** to visit each row in the result set

SQLite Simple Queries - Template Based Queries

- Simple SQLite queries use a *template* oriented schema whose goal is to 'help' non-SQL developers in their process of querying a database.
- This *template* exposes all the components of a basic SQL-select statement.
- Simple queries can *only* retrieve data from a *single table*.
- The method's signature has a fixed sequence of seven arguments representing:
 1. the table name,
 2. the columns to be retrieved,
 3. the search condition (where-clause),
 4. arguments for the where-clause,
 5. the group-by clause,
 6. having-clause, and
 7. the order-by clause.

SQLite Simple Queries - Template Based Queries

The signature of the SQLite simple **.query** method is:

```
db.query (table: String!,  
          columns: Array<out String!>!,  
          selection: String!,  
          selectionArgs: Array<out String!>!,  
          groupBy: String!,  
          having: String!,  
          orderBy: String!)
```

Example6. SQLite Simple Queries

In this example we use the **tblAmigo** table. We are interested in selecting the columns: *recID*, *name*, and *phone*. The condition to be met is that RecID must be greater than 2, and names must begin with 'B' and have three or more letters.

```
val columns = arrayOf("recID", "name", "phone")
val cs = db.query("tblAMIGO", columns,
    "recID > 2 and length(name) >= 3 and name like 'B%",
    null, null, null,
    "recID")
val numRecords = cs.count
```

We enter **null** in each component not supplied to the method. For instance, in this example select-args, having, and group-by are not used.

Example7. SQLite Simple Queries

In this example we will construct a more complex SQL select statement.

We are interested in tallying how many groups of friends whose recID > 3 have the same name. In addition, we want to see 'name' groups having no more than four people each.

A possible SQL-select statement for this query would be something like:

```
select  name, count(*) as TotalSubGroup
  from  tblAMIGO
 where  recID > 3
 group  by name
having  count(*) <= 4;
```

Example7. SQLite Simple Queries

An equivalent Android-SQLite solution using a simple template query follows.

```
val selectColumns = arrayOf("name", "count(*) as TotalSubGroup")
val whereCondition = "recID > ?"
val whereConditionArgs = arrayOf("3")
val groupBy = "name"
val having = "count(*) < 4"
val orderBy = "name"

val cs = db.query("tblAMIGO", selectColumns, whereCondition, whereConditionArgs,
    groupBy, having, orderBy)
```


Example7. SQLite Simple Queries

Observations

1. The *selectColumns* string array contains the output fields. One of them (*name*) is already part of the table, while *TotalSubGroup* is an alias for the computed count of each name sub-group.
2. The symbol **?** in the *whereCondition* is a *place-marker* for a substitution. The value “3” taken from the *whereConditionArgs* is to be injected there.
3. The *groupBy* clause uses ‘*name*’ as a key to create sub-groups of rows with the same *name* value. The *having* clause makes sure we only choose subgroups no larger than four people.

SQL Action Queries

Action queries are the SQL way of performing maintenance operations on tables and database resources. Example of action-queries include: *insert*, *delete*, *update*, *create table*, *drop*, etc.

Examples:

```
insert into tblAmigos  
  values ( 'Macarena', '555-1234' );
```

```
update tblAmigos  
  set name = 'Maria Macarena'  
  where phone = '555-1234';
```

```
delete from tblAmigos  
  where phone = '555-1234';
```

```
create table Temp ( column1 int, column2 text, column3 date );
```

```
drop table Temp;
```

SQLite Action Queries Using: ExecSQL

Perhaps the simplest Android way to phrase a SQL action query is to ‘stitch’ together the pieces of the SQL statement and give it to the easy to use –but rather limited- ***execSQL(...)*** method.

Unfortunately SQLite ***execSQL*** does **NOT** return any data. Therefore knowing how many records were affected by the action is not possible with this operator. Instead you should use the Android versions describe in the next section.

Android's INSERT, DELETE, UPDATE Operators

- Android provides a number of additional methods to perform *insert*, *delete*, *update* operations.
- They all return some feedback data such as the record ID of a recently inserted row, or number of records affected by the action. This format is recommended as a better alternative than execSQL.

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

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

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

ContentValues Class

- This class is used to store a set of **[name, value]** pairs (functionally equivalent to Bundles).
- When used in combination with SQLite, a ContentValues object is just a convenient way of passing a variable number of parameters to the SQLite action functions.
- Like bundles, this class supports a group of put/get methods to move data in/out of the container.

```
val myArgs= ContentValues()  
  
myArgs.put("name", "ABC")  
myArgs.put("phone", "555-7777")
```

myArgs

Key	Value
name	ABC
phone	555-7777

Android's INSERT Operation

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

The method tries to insert a row in a table. The row's column-values are supplied in the map called *values*. If successful, the method returns the **rowID** given to the new record, otherwise -1 is sent back.

Parameters

table	the table on which data is to be inserted
nullColumnHack	Empty and Null are different things. For instance, <i>values</i> could be defined but empty. If the row to be inserted <i>is empty</i> (as in our next example) this column will explicitly be assigned a NULL value (which is OK for the insertion to proceed).
values	Similar to a bundle (<i>name, value</i>) containing the column values for the row that is to be inserted.

Android's INSERT Operation

```
val rowValues = ContentValues()

rowValues.put("name", "ABC")
rowValues.put("phone", "555-1010")
var rowPosition = db.insert("tblAMIGO", null, rowValues)

rowValues.put("name", "DEF")
rowValues.put("phone", "555-2020")
rowPosition = db.insert("tblAMIGO", null, rowValues)

rowValues.clear()

rowPosition = db.insert("tblAMIGO", null, rowValues)

rowPosition = db.insert("tblAMIGO", "name", rowValues)
```

Android's INSERT Operation

Comments

1. A set of **<key, values>** called **rowValues** is created and supplied to the insert() method to be added to tblAmigo. Each *tblAmigo* row consists of the columns: *recID*, *name*, *phone*. Remember that *recID* is an *auto-incremented* field, its actual value is to be determined later by the database when the record is accepted.
2. The newly inserted record returns its rowID (4 in this example)
3. A second record is assembled and sent to the insert() method for insertion in tblAmigo. After it is collocated, it returns its rowID (5 in this example).
4. The rowValues map is reset, therefore rowValues which is not null becomes empty.
5. SQLite rejects attempts to insert an empty record returning rowID -1.
6. The second argument identifies a column in the database that allows NULL values (**NAME** in this case). Now SQL purposely inserts a NULL value on that column (as well as in other fields, except the key **RecId**) and the insertion successfully completes.

Android's UPDATE Operation

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

The method tries to update row(s) in a table. The SQL **set column=newvalue** clause is supplied in the *values* map in the form of [key,value] pairs. The method returns the number of records affected by the action.

Parameters

table	the table on which data is to be updated
values	Similar to a bundle (<i>name, value</i>) containing the columnName and NewValue for the fields in a row that need to be updated.
whereClause	This is the condition identifying the rows to be updated. For instance "name = ? " where ? Is a placeholder. Passing null updates the entire table.
whereArgs	Data to replace ? placeholders defined in the whereClause.

Android's UPDATE Operation

Example

We want to use the `.update()` method to express the following SQL statement:

```
update tblAmigo set name = 'maria' where (recID > 2 and recID < 7)
```

Here are the steps to make the call using Android's equivalent Update Method

```
val whereArgs = arrayOf("2", "7")
val updateValues = ContentValues()
updateValues.put("name", "Maria")
val recAffected = db.update("tblAMIGO", updateValues,
    "recID > ? and recID < ?", whereArgs)
```

Android's UPDATE Operation

Comments

1. Our **whereArgs** is an array of arguments. Those actual values will replace the placeholders '?' set in the whereClause.
2. The map **updValues** is defined and populated. In our case, once a record is selected for modifications, its "name" field will be changed to the new value "maria".
3. The **db.update()** method attempts to update all records in the given table that satisfy the filtering condition set by the **whereClause**. After completion it returns the number of records affected by the update (0 if it fails).
4. The update **filter** verifies that "*recID > ? and recID < ?*". After the args substitutions are made the new filter becomes: "*recID > 2 and recID < 7*".

Android's DELETE Operation

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

The method is called to delete rows in a table. A filtering condition and its arguments are supplied in the call. The condition identifies the rows to be deleted. The method returns the number of records affected by the action.

Parameters

table	the table on which data is to be deleted
whereClause	This is the condition identifying the records to be deleted. For instance "name = ? " where ? Is a placeholder. Passing null deletes all the rows in the table.
whereArgs	Data to replace '?' placeholders defined in the whereClause.

Android's DELETE Operation

Example

Consider the following SQL statement:

```
delete from tblAmigo where recID > 2 and recID < 7
```

An equivalent implementation using the Android's **delete method** follows:

```
val whereArgs = arrayOf("2", "7")  
val recAffected = db.delete("tblAMIGO",  
    "recID > ? and recID < ?", whereArgs)
```

A record should be deleted if its recID is in between the values 2, and 7. The actual values are taken from the *whereArgs* array. The method returns the number of rows removed after executing the command (or 0 if none).