Lesson 9

Persistence:
Files & Preferences
SQL Databases

Files & Preferences

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.

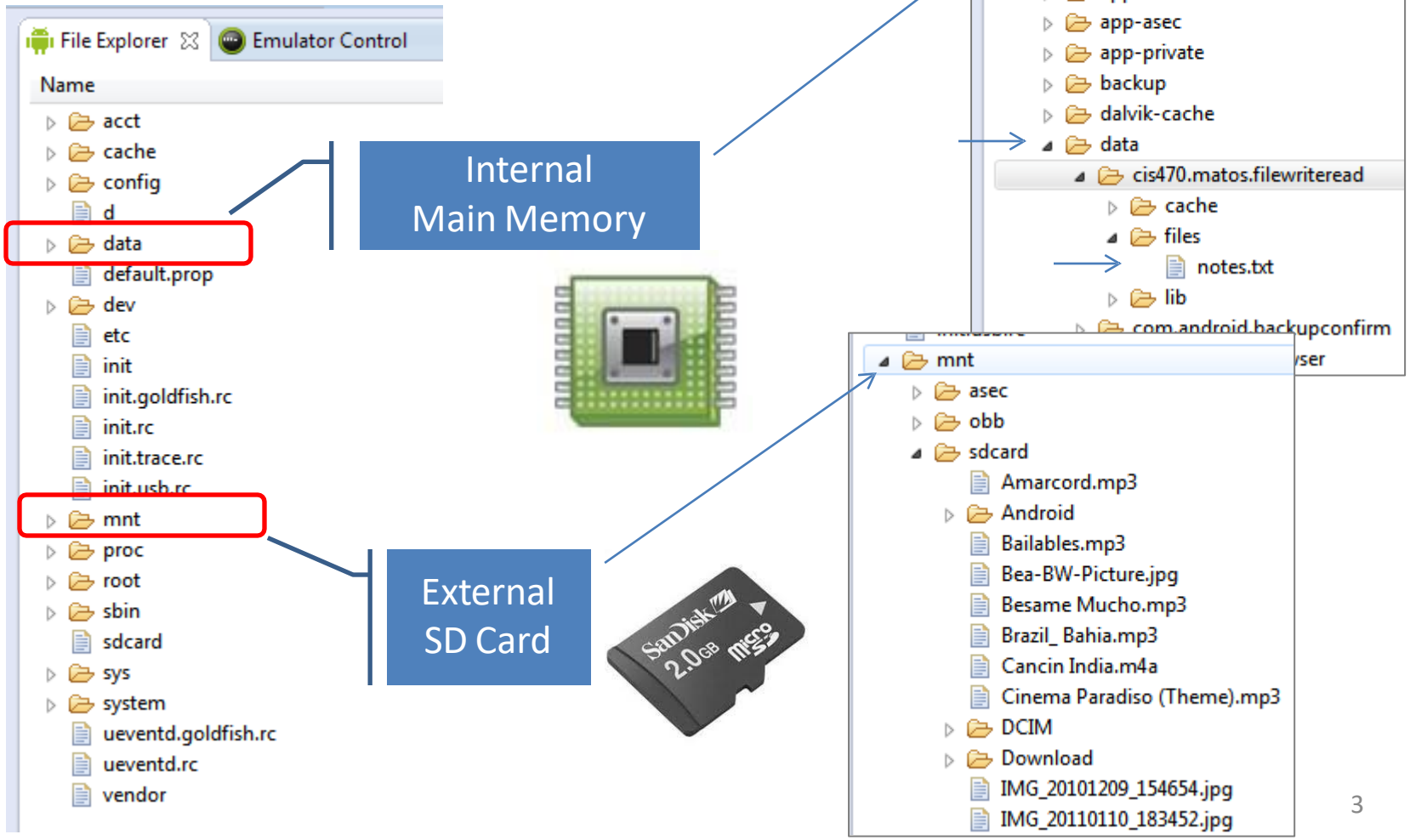
Files stored in the device's memory, share space with other application's resources such as code, icons, pictures, music, etc.

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

Files & Preferences

Exploring Android's File System

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



Files & Preferences

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.

Files & Preferences

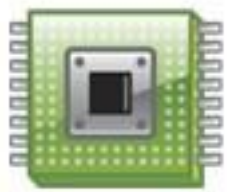
Shared Preferences

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.

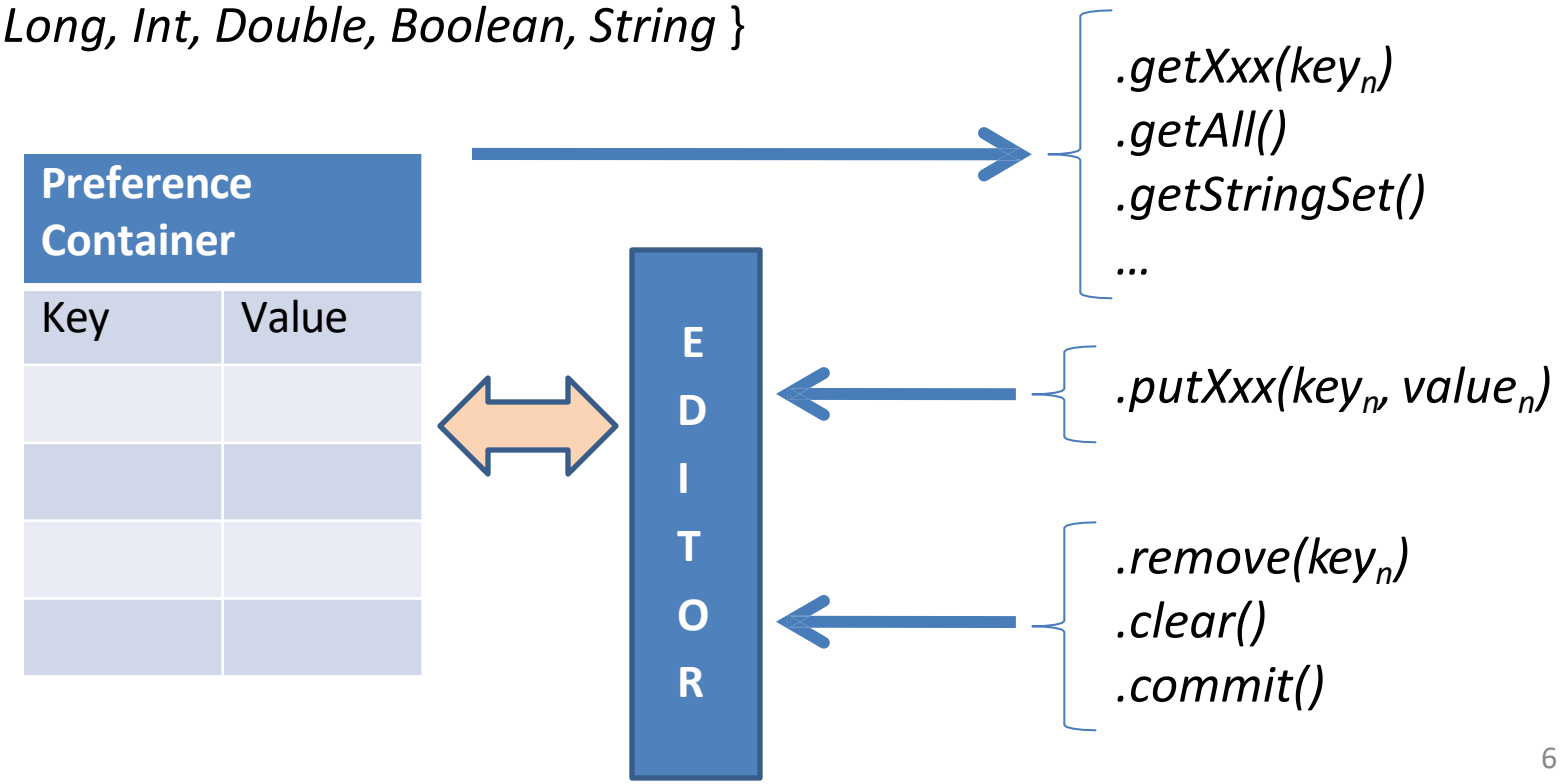
Files & Preferences

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* }



Files & Preferences

Example. Shared Preferences

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

Key	Value
chosenColor	RED
chosenNumber	7

```
private void usingPreferences(){
  // Save data in a SharedPreferences container
  // We need an Editor object to make preference changes.

  SharedPreferences myPrefs = getSharedPreferences("my_preferred_choices",
                                                   Activity.MODE_PRIVATE);
  SharedPreferences.Editor editor = myPrefs.edit();
          editor.putString("chosenColor", "RED");
          editor.putInt("chosenNumber", 7 );
  editor.commit();

  // retrieving data from SharedPreferences container (apply default if needed)
  String favoriteColor = myPrefs.getString("chosenColor", "BLACK");
  int favoriteNumber = myPrefs.getInt("chosenNumber", 11 );

}
```

1 → `SharedPreferences myPrefs = getSharedPreferences("my_preferred_choices", Activity.MODE_PRIVATE);`

2 → `editor.putString("chosenColor", "RED");` `editor.putInt("chosenNumber", 7 );`

3 → `String favoriteColor = myPrefs.getString("chosenColor", "BLACK");` `int favoriteNumber = myPrefs.getInt("chosenNumber", 11 );`

Files & Preferences

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".

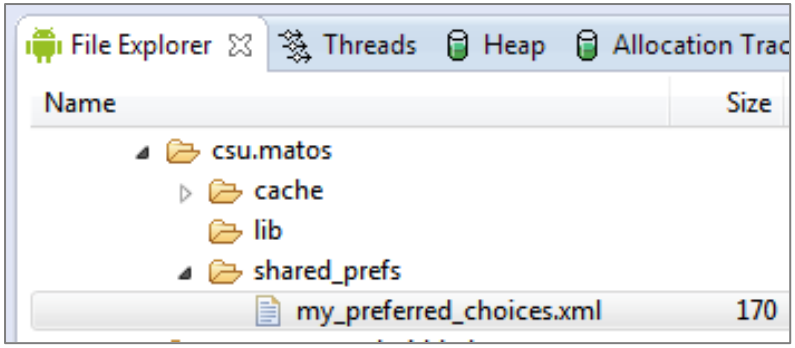
Files & Preferences

Shared Preferences. Example - Comments

SharedPreference 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

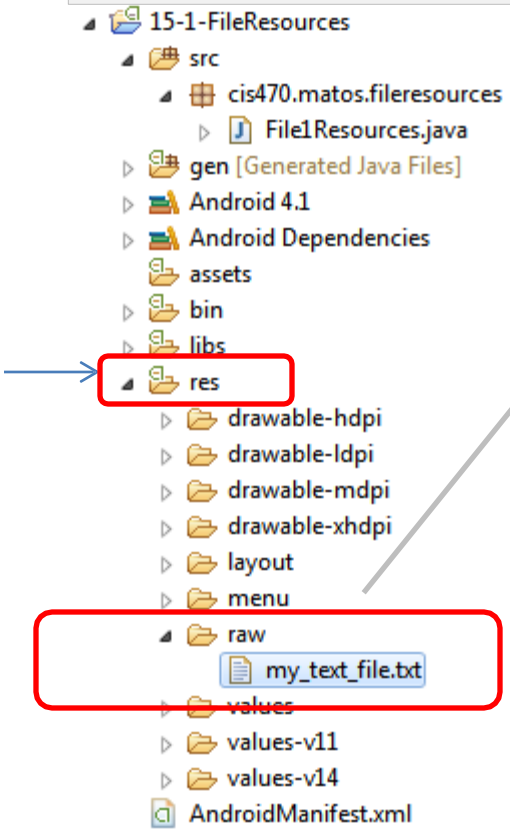
Internal Storage. Reading an Internal Resource File

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

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

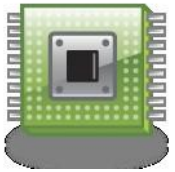
Resources could be accessed through the **.getResources(…)** method. The method's argument is the ID assigned by Android to the element in the R resource file. For example:

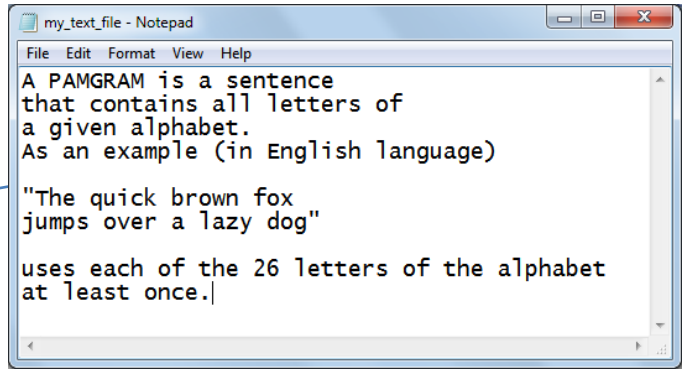
```
InputStream is = this.getResources()
                        .openRawResource(R.raw.my_text_file);
```



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



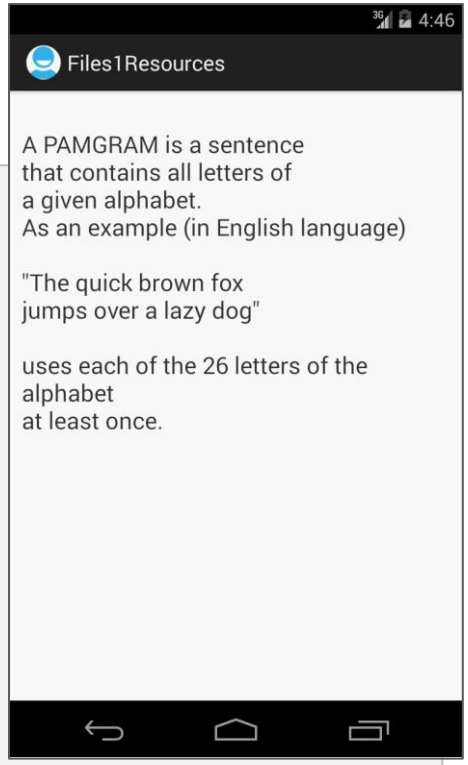


Example of a pamgram in Spanish:
La cigüeña tocaba cada vez mejor el saxofón y el búho pedía whiskey y queso.

Files & Preferences

Example 1. Reading an Internal Resource File

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




```
//reading an embedded RAW data file
public class File1Resources extends Activity {
   TextView txtMsg;
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);

      txtMsg = (TextView) findViewById(R.id.textView1);
      try {
         PlayWithRawFiles();

      } catch (IOException e) {
         txtMsg.setText( "Problems: " + e.getMessage() );
      }
   }//onCreate
```

Files & Preferences

Example 1. Reading an Internal Resource File 2 of 2

Reading an embedded file containing lines of text.

```
public void PlayWithRawFiles() throws IOException {
   String str="";
   StringBuffer buf = new StringBuffer();

1  int fileResourceId = R.raw.my_text_file;
   InputStream is = this.getResources().openRawResource(fileResourceId);
2  BufferedReader reader = new BufferedReader(new
                                 InputStreamReader(is) );

   if (is!=null) {
     while ((str = reader.readLine()) != null) {
3        buf.append(str + "\n" );
         }
   }
   reader.close();
   is.close();
   txtMsg.setText( buf.toString() );

  }// PlayWithRawFiles
} // File1Resources
```

Files & Preferences

Example1 - Comments

1. A **raw file** is an arbitrary dataset stored in its original raw format (such as .docx, pdf, gif, jpeg, etc). Raw files can be accessed through an *InputStream* acting on a *R.raw.filename* resource entity.
   CAUTION: *Android requires resource file names to be in lowercase form.*

2. The expression `getResources().openRawResource(fileResourceId)` creates an InputStream object that sends the bytes from the selected resource file to an input buffer. If the resource file is not found it raises a *NotFoundException* condition.

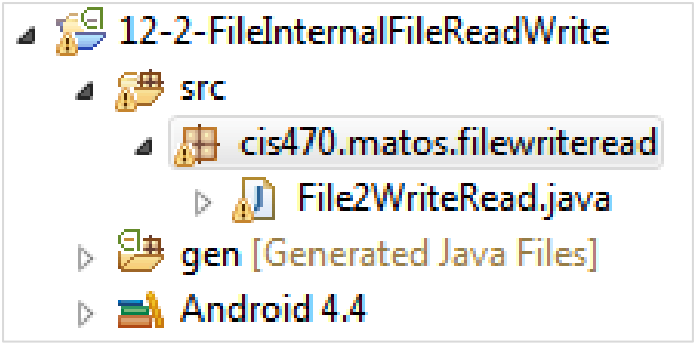
3. A *BufferedReader* object is responsible for extracting lines from the input buffer and assembling a string which finally will be shown to the user in a textbox. Protocol expects that conventional IO housekeeping operations should be issued to close the reader and stream objects.

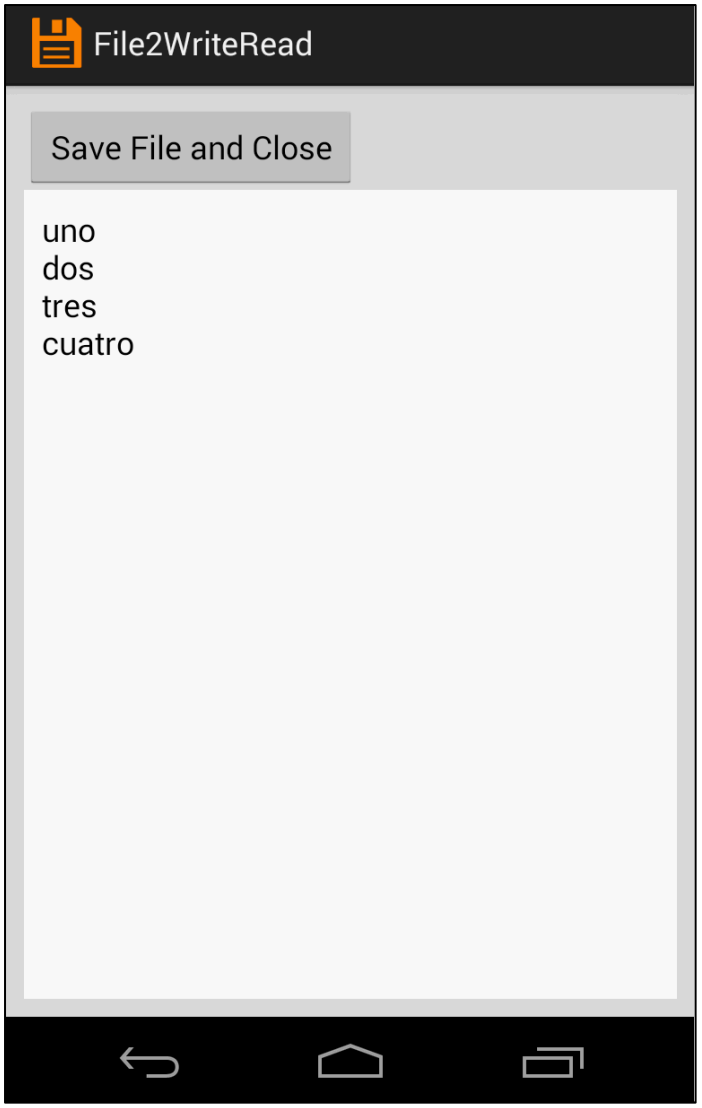
Files & Preferences

Example 2. Reading /Writing an Internal Resource File 1 of 6

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 *Resource File* will be **read** and its data will be shown on the UI.



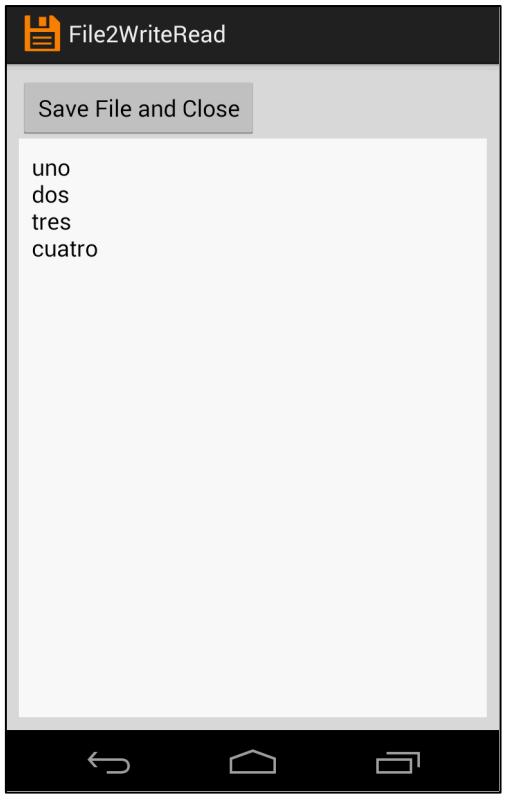


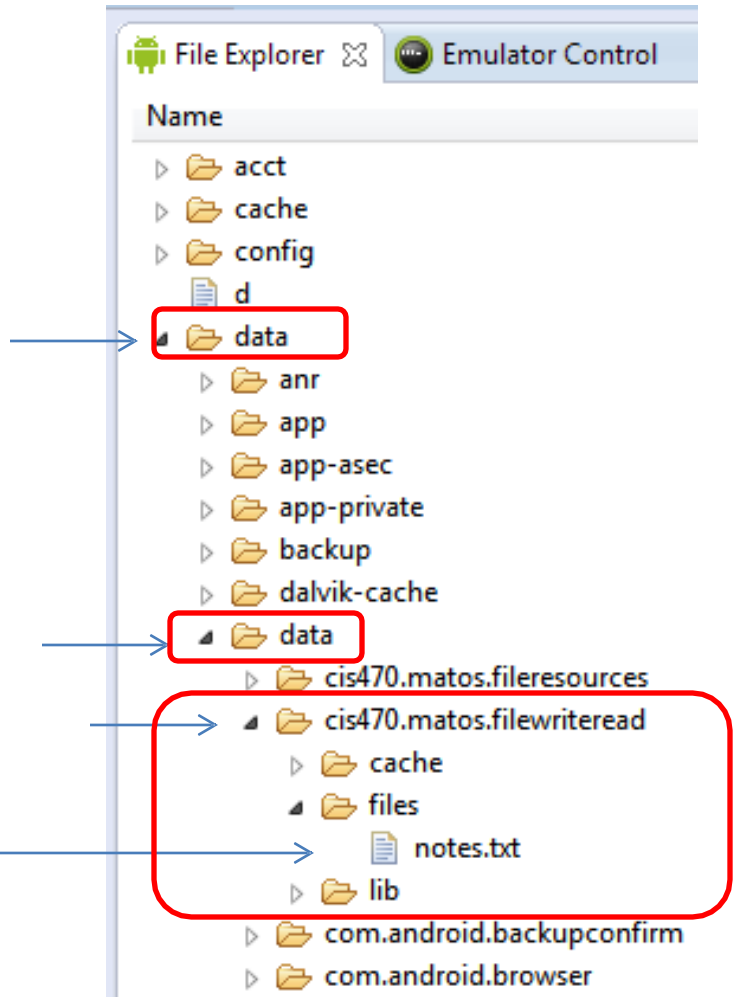
Files & Preferences

Example 2. Reading /Writing an Internal Resource File 2 of 6

The *internal resource file* (notes.txt) is private and cannot be seen by other apps residing in main memory.

In our example the files **notes.txt** is stored in the phone's internal memory under the name:
/data/data/cis470.matos.fileresources/files/notes.txt

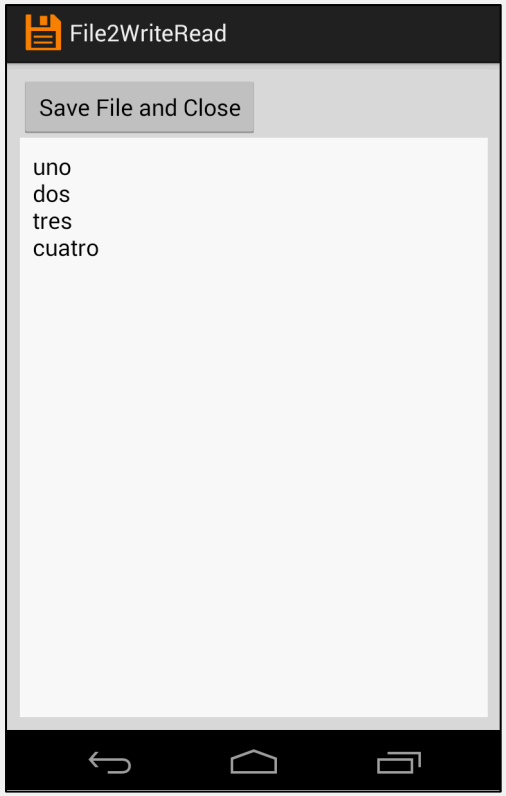




Files & Preferences

Example 2. Reading /Writing an Internal Resource File 3 of 6

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#ffdddddd"
    android:padding="10dp"
    android:orientation="vertical" >

    <Button android:id="@+id/btnFinish"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="10dp"
        android:text=" Save File and Close " />

    <EditText
        android:id="@+id/txtMsg"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="10dp"
        android:background="#ffffffff"
        android:gravity="top"
        android:hint="Enter some lines of data here..."  />

</LinearLayout>
```



Files & Preferences

Example 2. Reading /Writing an Internal Resource File 4 of 6

```
public class File2WriteRead extends Activity {

   private final static String FILE_NAME = "notes.txt";
   private EditText txtMsg;

   @Override
   public void onCreate(Bundle icicle) {
      super.onCreate(icicle);
      setContentView(R.layout.main);
      txtMsg = (EditText) findViewById(R.id.txtMsg);

      // deleteFile();  //keep for debugging

      Button btnFinish = (Button) findViewById(R.id.btnFinish);
      btnFinish.setOnClickListener(new Button.OnClickListener() {
         public void onClick(View v) {
            finish();
         }
      });

   }// onCreate
```

Files & Preferences

Example 2. Reading /Writing an Internal Resource File 5 of 6

```
public void onStart() {
  super.onStart();

  try {
    InputStream inputStream = openFileInput(FILE_NAME);

    if (inputStream != null) {

      BufferedReader reader = new BufferedReader(new
                                  InputStreamReader(inputStream));
      String str = "";
      StringBuffer stringBuffer = new StringBuffer();

      while ((str = reader.readLine()) != null) {
        stringBuffer.append(str + "\n");
      }

      inputStream.close();
      txtMsg.setText(stringBuffer.toString());
    }
  }
  catch ( Exception ex ) {
    Toast.makeText(CONTEXT, ex.getMessage() , 1).show();
  }
}// onStart
```

Files & Preferences

Example 2. Reading /Writing an Internal Resource File 6 of 6

```
public void onPause() {
   super.onPause();
   try {
      OutputStreamWriter out = new OutputStreamWriter(
3 →                                   openFileOutput(FILE_NAME, 0));
      out.write(txtMsg.getText().toString());
      out.close();
   } catch (Throwable t) {
      txtMsg.setText( t.getMessage() );
   }
}// onPause
```

```
private void deleteFile() {
   String path = "/data/data/cis470.matos.filewriteread/files/" + FILE_NAME;
   File f1 = new File(path);
   Toast.makeText(getApplicationContext(), "Exists?" + f1.exists() , 1).show();
4 →  boolean success = f1.delete();
   if (!success){
     Toast.makeText(getApplicationContext(), "Delete op. failed.", 1).show();
   }else{
      Toast.makeText(getApplicationContext(), "File deleted.", 1).show();
   }
}
```

Files & Preferences

Example2 - Comments

1. The expression **`openFileInput`**`(`*`FILE_NAME`*`)` opens a private file linked to this *Context's* application package for reading. This is an alternative to the method `getResources().openRawResource(fileResourceId)` discussed in the previous example.

2. A *BufferedReader* object moves data line by line from the input file to a textbox. After the buffer is emptied the data sources are closed.

3. An *OutputStreamWriter* takes the data entered by the user and send this stream to an internal file. The method **`openFileOutput()`** opens a private file for writing and creates the file if it doesn't already exist. The file's path is: **/data/data/packageName/FileName**

4. You may delete an existing resource file using conventional `.delete()` method.

Files & Preferences

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.

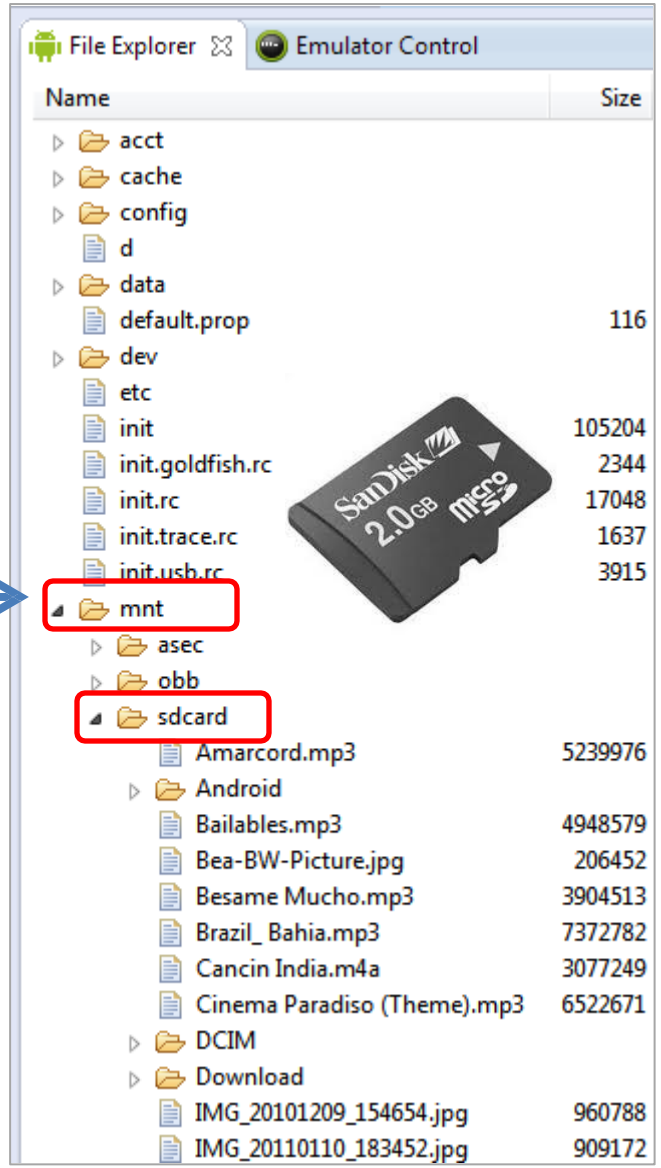
SD cards are ideal for keeping your collection of music, picture, ebooks, and video files.

Files & Preferences

Reading /Writing External SD Files

Use the **File Explorer** tool to locate files in your device (or emulator).

Look into the folder: **mnt/sdcard/** there you typically keep music, pictures, videos, etc.



Files & Preferences

Reading /Writing External SD Files

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

mnt/sdcard/mysdfile.txt

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

```
String sdPath = Environment.getExternalStorageDirectory().getAbsolutePath() ;
```

WARNING

When you deal with external files you need to request permission to read and write to the SD card. Add the following clauses to your AndroidManifest.xml

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

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

Files & Preferences

Reading /Writing External SD Files

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

```
if (checkSelfPermission(Manifest.permission.WRITE_EXTERNAL_STORAGE)
    != PackageManager.PERMISSION_GRANTED) {
    requestPermissions(
        new String[] {Manifest.permission.WRITE_EXTERNAL_STORAGE},
        REQUEST_CODE);
}
```

```
@Override
public void onRequestPermissionsResult(int requestCode, String[]
permissions, int[] grantResults) {
    if (requestCode == 1234)
        if (grantResults[0] != PackageManager.PERMISSION_GRANTED)
            Log.v("TAG", "Permission Denied");
}
```

Files & Preferences

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 >= 30 && !Environment.isExternalStorageManager()) {
   Uri uri = Uri.parse("package:" + BuildConfig.APPLICATION_ID);
   Intent intent = new Intent(Settings.ACTION_MANAGE_APP_ALL_FILES_ACCESS_PERMISSION, uri);
   startActivity(intent);
}
```

Files & Preferences

Reading /Writing External SD Files

Detect path for emulated and real SD card

```
if (Build.VERSION.SDK_INT >= 19) {
    File files[] = getExternalFilesDirs(null);
    for (File f : files) {
        String path = f.getAbsolutePath();
        Log.v("TAG", path);
    }
}
```

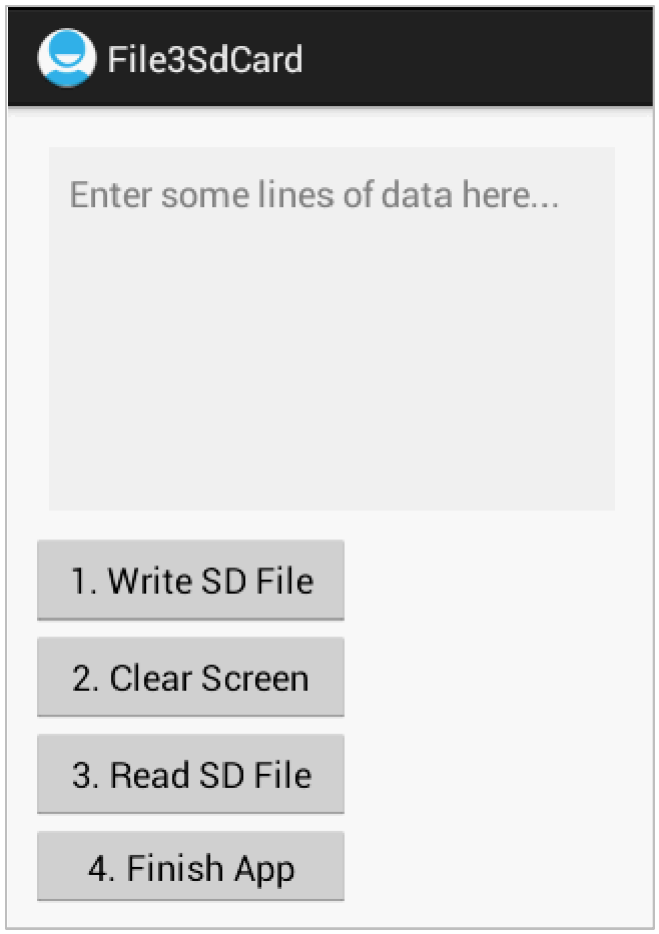
Result in logcat:

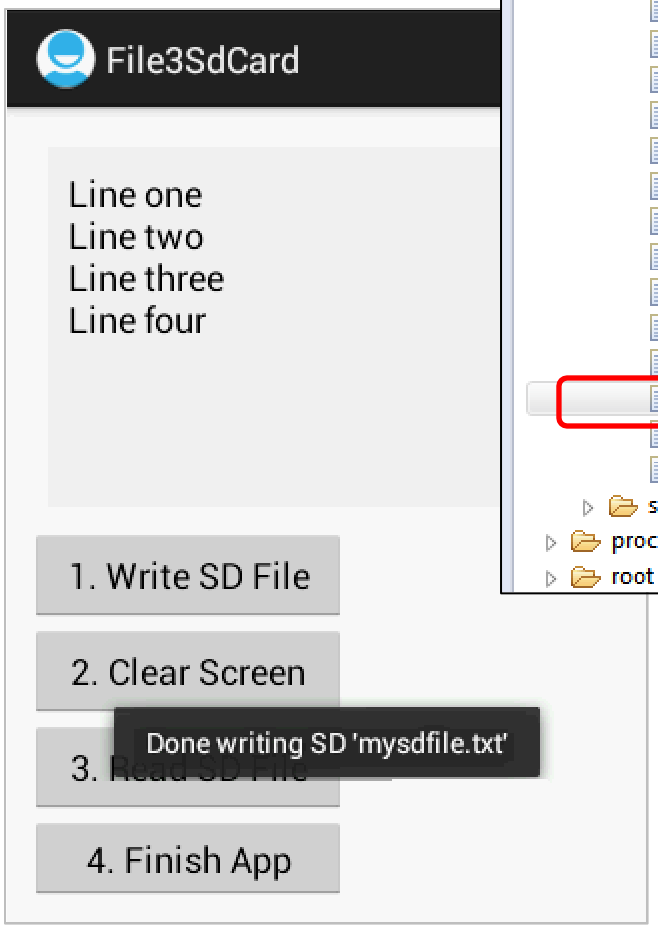
/storage/emulated/0/Android/data/vn.edu.hust.myapplication/files
/storage/4CFC-8D04/Android/data/vn.edu.hust.myapplication/files

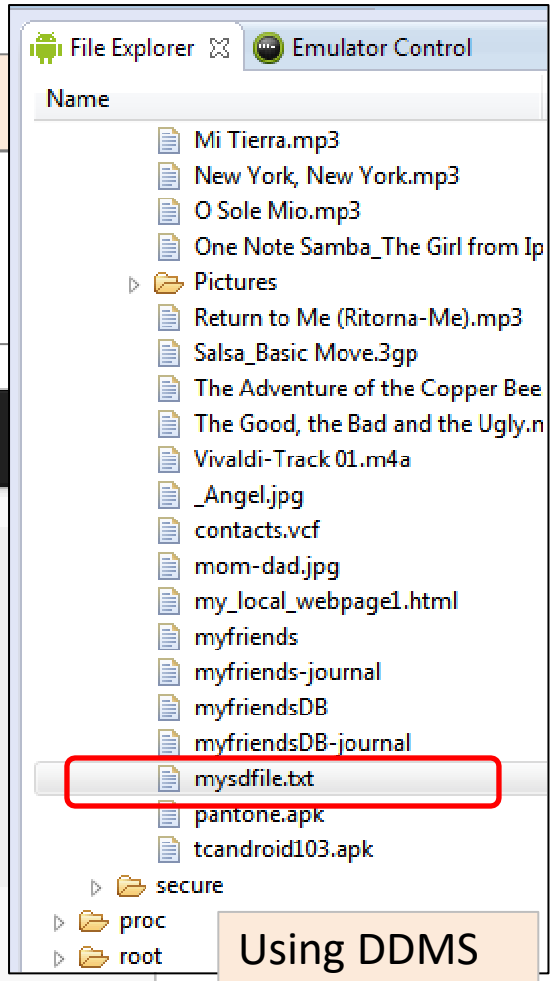
Files & Preferences

Example 3. Reading /Writing External SD Files

This app accepts a few lines of user input and writes it to the external SD card. User clicks on buttons to either have the data read and brought back, or terminate the app.



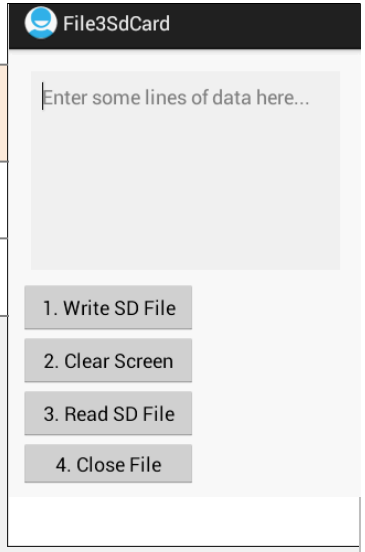




Using DDMS **File Explorer** to inspect the SD card.

Files & Preferences

Example 3. Reading /Writing External SD Files



Layout

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/widget28"
    android:padding="10dp"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <EditText
        android:id="@+id/txtData"
        android:layout_width="match_parent"
        android:layout_height="180dp"
        android:layout_margin="10dp"
        android:background="#55dddddd"
        android:padding="10dp"
        android:gravity="top"
        android:hint=
       "Enter some lines of data here..."
        android:textSize="18sp" />
    <Button
        android:id="@+id/btnWriteSDFile"
        android:layout_width="160dp"
        android:layout_height="wrap_content"
        android:text="1. Write SD File" />
```

```
    <Button
        android:id="@+id/btnClearScreen"
        android:layout_width="160dp"
        android:layout_height="wrap_content"
        android:text="2. Clear Screen" />

     <Button
        android:id="@+id/btnReadSDFile"
        android:layout_width="160dp"
        android:layout_height="wrap_content"
        android:text="3. Read SD File" />

     <Button
        android:id="@+id/btnFinish"
        android:layout_width="160dp"
        android:layout_height="wrap_content"
        android:text="4. Finish App" />

</LinearLayout>
```

Files & Preferences

Example 3. Reading /Writing External SD Files


```
public class File3SdCard extends Activity {
   // GUI controls
   private EditText txtData;
   private Button btnWriteSDFile;
   private Button btnReadSDFile;
   private Button btnClearScreen;
   private Button btnClose;
   private String mySdPath;

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);

      // find SD card absolute location
1 →   mySdPath = Environment.getExternalStorageDirectory().getAbsolutePath();

      // bind GUI elements to local controls
      txtData = (EditText) findViewById(R.id.txtData);
      txtData.setHint("Enter some lines of data here...");
```

Files & Preferences

Example 3. Reading /Writing External SD Files


```
btnWriteSDFile = (Button) findViewById(R.id.btnWriteSDFile);
btnWriteSDFile.setOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View v) {
    // WRITE on SD card file data taken from the text box
    try {
      File myFile = new File(mySdPath + "/mysdfile.txt");

      OutputStreamWriter myOutWriter = new OutputStreamWriter(
                                        new FileOutputStream(myFile));

      myOutWriter.append(txtData.getText());
      myOutWriter.close();

      Toast.makeText(getBaseContext(),
          "Done writing SD 'mysdfile.txt'",
          Toast.LENGTH_SHORT).show();
    } catch (Exception e) {
      Toast.makeText(getBaseContext(), e.getMessage(),
          Toast.LENGTH_SHORT).show();
    }
  }// onClick
}); // btnWriteSDFile
```

2

Files & Preferences

Example 3. Reading /Writing External SD Files


```
btnReadSDFile = (Button) findViewById(R.id.btnReadSDFile);
btnReadSDFile.setOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View v) {
    // READ data from SD card show it in the text box
    try {
      BufferedReader myReader = new BufferedReader(
3 →                                 new InputStreamReader(
                                    new FileInputStream(
                                    new File(mySdPath + "/mysdfile.txt"))));
      String aDataRow = "";
      String aBuffer = "";
      while ((aDataRow = myReader.readLine()) != null) {
        aBuffer += aDataRow + "\n";
      }
      txtData.setText(aBuffer);
      myReader.close();
      Toast.makeText(getApplicationContext(),
          "Done reading SD 'mysdfile.txt'", Toast.LENGTH_SHORT).show();
    } catch (Exception e) {
      Toast.makeText(getApplicationContext(), e.getMessage(),
          Toast.LENGTH_SHORT).show();
    }
  }// onClick
}); // btnReadSDFile
```

Files & Preferences

Example 3. Reading /Writing External SD Files


```
      btnClearScreen = (Button) findViewById(R.id.btnClearScreen);
      btnClearScreen.setOnClickListener(new OnClickListener() {
         @Override
         public void onClick(View v) {
            // clear text box
            txtData.setText("");
         }
      }); // btnClearScreen

      btnClose = (Button) findViewById(R.id.btnFinish);
      btnClose.setOnClickListener(new OnClickListener() {
         @Override
         public void onClick(View v) {
            // terminate app
            Toast.makeText(getApplicationContext(),
                  "Adios...", Toast.LENGTH_SHORT).show();
            finish();
         }
      }); // btnClose

   }// onCreate

}// File3SdCard
```

SQL Databases

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.

SQL Databases

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

SQL Databases

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 *SQLiteException* if the database cannot be opened

SQL Databases

Example1: Creating a SQLite database - Method 1

```
package cis470.matos.sqldatabases;
public class MainActivity extends Activity {
  SQLiteDatabase db;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    TextView txtMsg = (TextView) findViewById(R.id.txtMsg);

    // path to the external SD card (something like: /storage/sdcard/...)
    // String storagePath = Environment.getExternalStorageDirectory().getPath();

    // path to internal memory file system (data/data/cis470.matos.databases)
    File storagePath = getApplication().getFilesDir();

    String myDbPath = storagePath + "/" + "myfriends";

    txtMsg.setText("DB Path: " + myDbPath);
    try {
      db = SQLiteDatabase.openDatabase(myDbPath, null,
                                       SQLiteDatabase.CREATE_IF_NECESSARY);
      // here you do something with your database ...
      db.close();
      txtMsg.append("\nAll done!");
    } catch (SQLiteException e) {
      txtMsg.append("\nERROR " + e.getMessage());
    }
  }// onCreate
}// class
```

SQL Databases

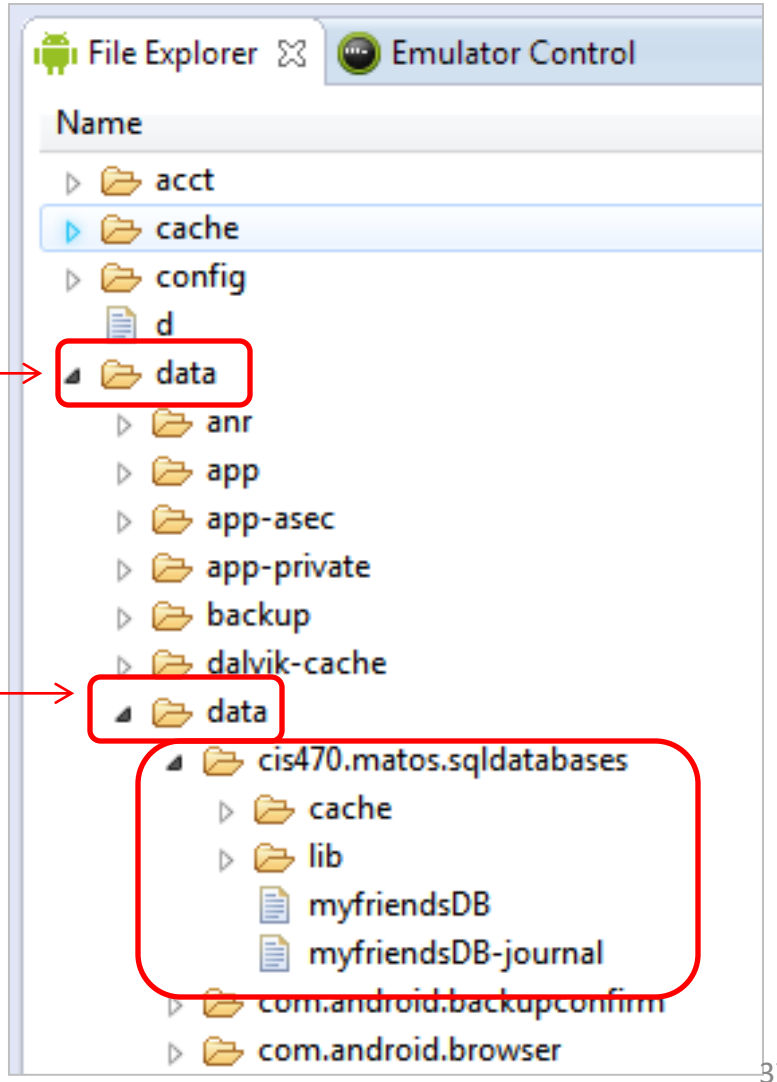
Example1: Creating a SQLite database - Using Memory

SQLite Database is stored using Internal Memory

Path:

/data/data/cis470.matos.sqldatabases/

Where: cis470.matos.sqldatabases
is the package's name



SQL Databases

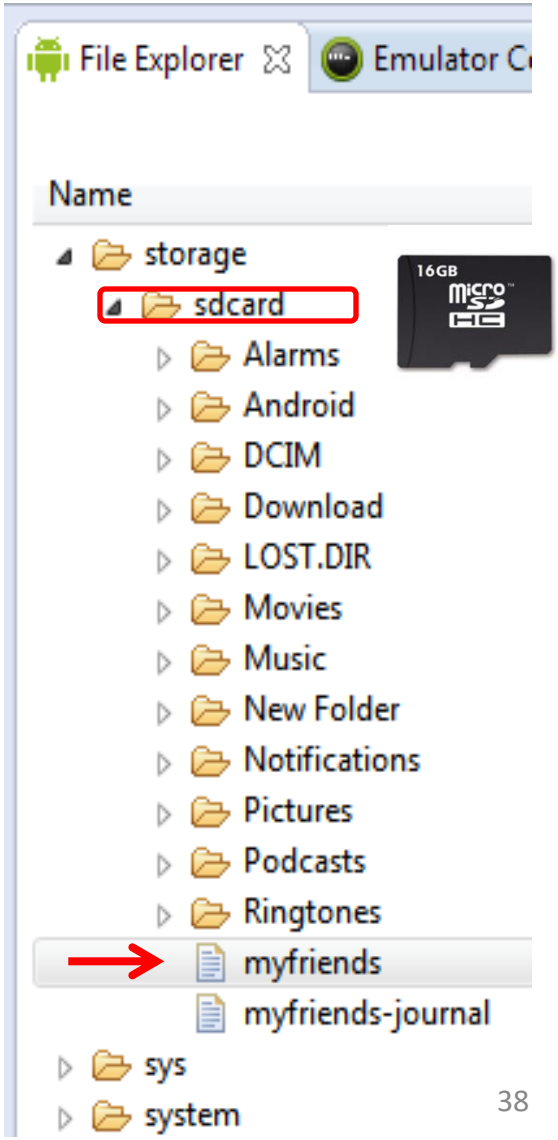
Example1: Creating a SQLite database on the SD card

Using:

```
SQLiteDatabase db;
String SDcardPath = Environment
            .getExternalStorageDirectory()
            .getPath() + "/myfriends";

db = SQLiteDatabase.openDatabase(
        SDcardPath,
        null,
        SQLiteDatabase.CREATE_IF_NECESSARY
        );
```

Manifest must include:

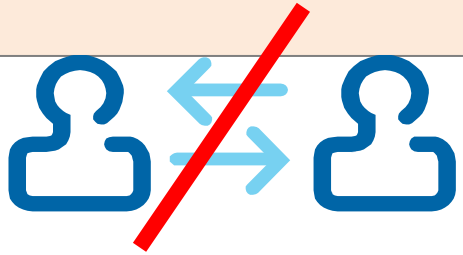
```
<uses-permission android:name=
"android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name=
"android.permission.READ_EXTERNAL_STORAGE" />
```



SQL Databases

Sharing Limitations

Warning



- Databases created in the internal **/data/data/package** space are private to that package.
- You *cannot* access internal databases belonging to other people (instead use Content Providers or external SD resident DBs).
- SD stored databases are *public*.
- Access to an SD resident database requires the Manifest to include permissions:

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

NOTE*: SQLITE (as well as most DBMSs) is not case sensitive.*

SQL Databases

An Alternative Method: openOrCreateDatabase

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

```
SQLiteDatabase  db =  this.openOrCreateDatabase(
                                    "myfriendsDB",
                                     MODE_PRIVATE,
                                     null);
```

Assume this app is made in a namespace called **cis470.matos.sqldatabases**, then the full name of the newly created database file will be:

/data/data/cis470.matos.sqldatabases/myfriendsDB

Internal Memory | Package name | DB name

- 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)

SQL Databases

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.

SQL Databases

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'.*

SQL Databases

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 (SQLiteException e) {
   //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*

SQL Databases

Create and Populate a SQL Table

recID	name	phone
1	AAA	555-1111
2	BBB	555-2222
3	CCC	555-3333

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.

SQL Databases

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.

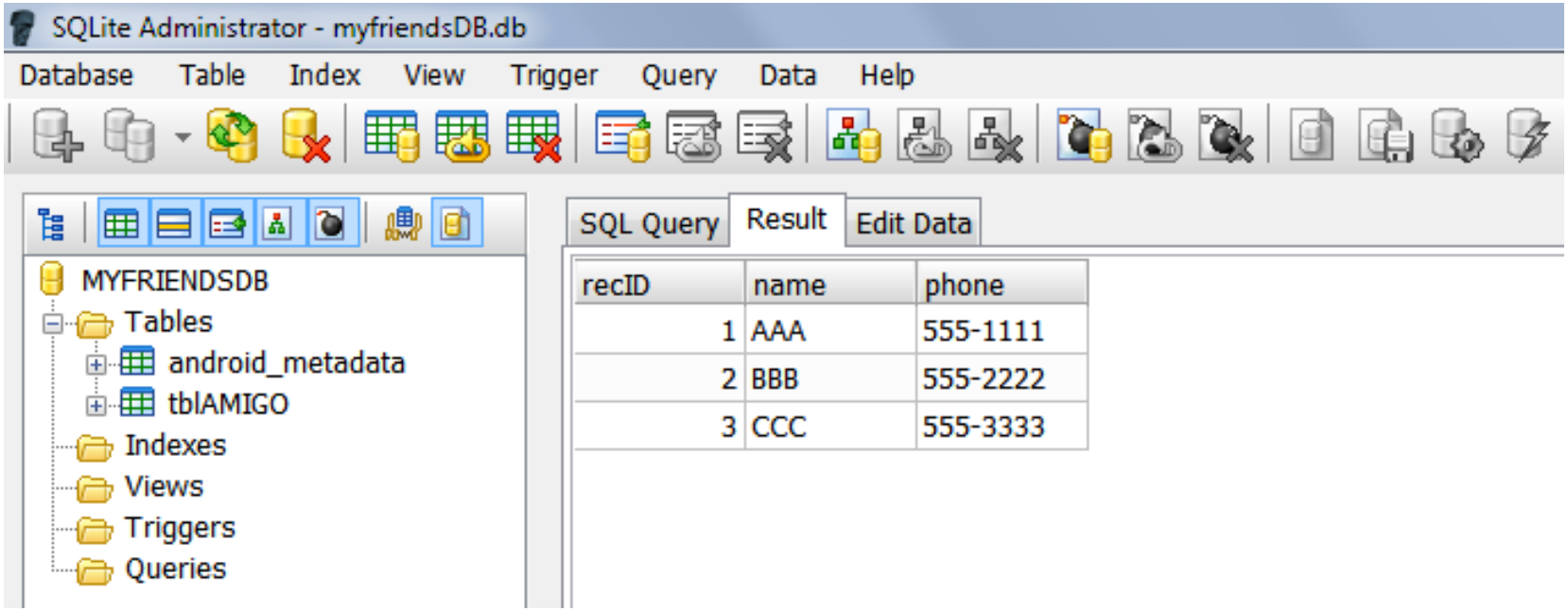
***Note**: For presentation economy we do not show the entire code which should include a transaction frame.*

```
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');"  );
```

SQL Databases

Example 2. Create and Populate a SQL Table

- After executing the previous code snippet, we transfered the database to the developer's file system and used the SQL-ADMINISTRATION tool.
- There we submitted the SQL-Query: **select * from tblAmigo**.
- Results are shown below.



SQL Databases

recID	name	phone
1	AAA	555
2	BBB	777
3	CCC	999

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.

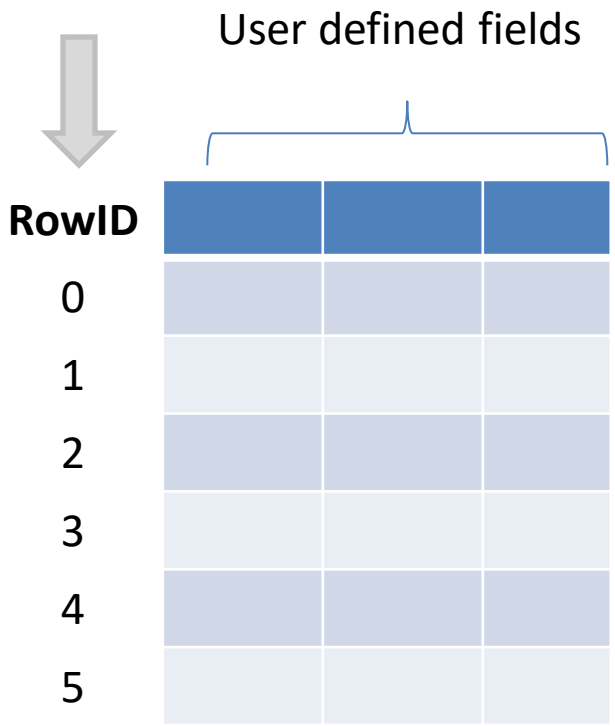
SQL Databases

Example 2. Create and Populate a SQL Table

NOTE:

SQLITE uses an **invisible** field called ***ROWID*** to uniquely identify each row in each table.

Consequently in our example the field ***recID*** and the database ***ROWID*** are functionally similar.



SQL Databases

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 Databases

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.

SQL Databases

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
```

SQL Databases

Example3. Using a Parameterless RawQuery (version 1)

Consider the following code fragment

```
Cursor c1 = 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).

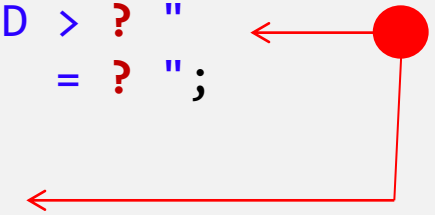
SQL Databases

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:

```
String mySQL = "select count(*) as Total "
                + " from tblAmigo "
                + " where recID > ? "
                + "   and name  = ? ";

 String[] args = {"1", "BBB"};

 Cursor c1 = db.rawQuery(mySQL, 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".

SQL Databases

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:

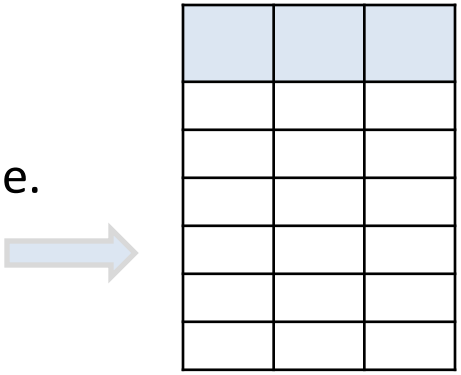
```
String[] args = {"1", "BBB"};

String mySQL = " select count(*) as Total "
               + "  from tblAmigo "
               + " where recID > "  + args[0]
               + "   and name  = '" + args[1] + "'";

Cursor c1 = db.rawQuery(mySQL, null);
```

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

SQL Databases

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().

SQL Databases

Example 4A. Traversing a Cursor – Simple Case


```
String sql = "select * from tblAmigo";
Cursor c1 = db.rawQuery(sql, null);

c1.moveToPosition(-1);

while ( c1.moveToNext() ){

   int recId = c1.getInt(0);
   String name = c1.getString(1);
   String phone = c1.getString(c1.getColumnIndex("phone"));

   // do something with the record here...

}
```

1. Prepare a rawQuery passing a simple sql statement with no arguments, catch the resulting tuples in cursor **c1**.
2. Move the fetch marker to the absolute position prior to the first row in the file. The valid range of values is -1 <= position <= count.
3. Use **moveToNext()** to visit each row in the result set

SQL Databases

Example 4B. Traversing a Cursor – Enhanced Navigation 1 of 2

```
1 → private String showCursor( Cursor cursor) {
      // reset cursor's top (before first row)
      cursor.moveToPosition(-1);
      String cursorData = "\nCursor: [";

      try {
         // get SCHEMA (column names & types)
2 →      String[] colName = cursor.getColumnNames();
         for(int i=0; i<colName.length; i++){
            String dataType = getColumnType(cursor, i);
            cursorData += colName[i] + dataType;

            if (i<colName.length-1){
               cursorData+= ", ";
            }
         }
      } catch (Exception e) {
         Log.e( "<<SCHEMA>>" , e.getMessage() );
      }
      cursorData += "]";

      // now get the rows
      cursor.moveToPosition(-1); //reset cursor's top
```

SQL Databases

Example 4B. Traversing a Cursor – Enhanced Navigation 2 of 2

```
3 →   while (cursor.moveToNext()) {
         String cursorRow = "\n[";
         for (int i = 0; i < cursor.getColumnCount(); i++) {
4 →         cursorRow += cursor.getString(i);
            if (i<cursor.getColumnCount()-1)
               cursorRow +=  ", ";
         }
         cursorData += cursorRow + "]";
      }
      return cursorData + "\n";
   }

   private String getColumnType(Cursor cursor, int i) {
5 →      try {
         //peek at a row holding valid data
         cursor.moveToFirst();
         int result = cursor.getType(i);
         String[] types = {":NULL", ":INT", ":FLOAT", ":STR", ":BLOB", ":UNK" };
         //backtrack - reset cursor's top
         cursor.moveToPosition(-1);
         return types[result];
      } catch (Exception e) {
         return " ";
      }
   }
```

SQL Databases

Comments Example 4B – Enhanced Navigation

1. The method: **showCursor(Cursor cursor)** implements the process of visiting individual rows retrieved by a SQL statement. The argument **cursor**, is a wrapper around the SQL resultset. For example, you may assume **cursor** was created using a statement such as:
```
Cursor cursor = db.rawQuery("select * from tblAMIGO", null);
```

2. The database **schema** for tblAmigo consists of the attributes: *recID*, *name*, and *phone*. The method *getColumnNames*() provides the schema.

3. The method *moveToNext* forces the cursor to travel from its current position to the next available row.

4. The accessor *.getString* is used as a convenient way of extracting SQL fields without paying much attention to the actual data type of the fields.

5. The function *.getColumnType()* provides the data type of the current field (0:null, 1:int, 2:float, 3:string, 4:blob)

SQL Databases

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.

SQL Databases

SQLite Simple Queries - Template Based Queries

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

```
db.query (String   table,
          String[] columns,
          String   selection,
          String[] selectionArgs,
          String   groupBy,
          String   having,
          String   orderBy )
```

SQL Databases

Example5. SQLite Simple Queries

Assume we need to consult an **EmployeeTable** (see next Figure) and find the average salary of female employees supervised by emp. 123456789. Each output row consists of Dept. No, and ladies-average-salary value. Our output should list the highest average first, then the second, and so on. Do not include depts. having less than two employees.

```
String[] columns = {"Dno", "Avg(Salary) as AVG"};

String[] conditionArgs = {"F", "123456789"};

Cursor c = db.query("EmployeeTable",
                    columns,
                    "sex = ? And superSsn = ? ",
                    conditionArgs,
                    "Dno",
                    "Count(*) > 2",
                    "AVG Desc "
                   );
```

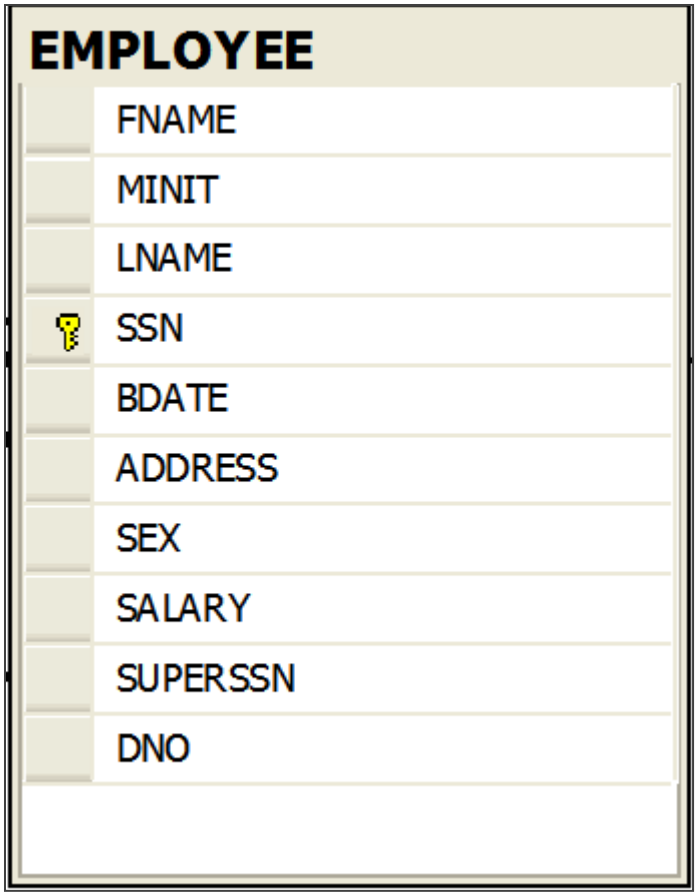
```
← table name
← ouput columns
← condition
← condition-args
← group by
← having
← order by
```

SQL Databases

Example5. SQLite Simple Queries

This is a representation of the **EmployeeTable** used in the previous example.

It contains: first name, initial, last name, SSN, birthdate, address, sex, salary, supervisor's SSN, and department number.



SQL Databases

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.

```
String [] columns = {"recID", "name", "phone"};

Cursor c1 = db.query (
       "tblAMIGO",
       columns,
       "recID > 2 and length(name) >= 3 and name like 'B%' ",
       null, null, null,
       "recID" );

int recRetrieved = c1.getCount();
```

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.

SQL Databases

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;
```

SQL Databases

Example7. SQLite Simple Queries

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

```
1 → String [] selectColumns =  {"name", "count(*) as TotalSubGroup"};
2 → String    whereCondition = "recID > ? ";
    String [] whereConditionArgs = {"3"};
3 → String    groupBy = "name";
    String    having =  "count(*) <= 4";
    String    orderBy = "name";

    Cursor cursor =  db.query (
                             "tblAMIGO",
                             selectColumns,
                             whereCondition,
                             whereConditionArgs,
                             groupBy,
                             having,
                             orederBy );
```

SQL Databases

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 Databases

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;
```

SQL Databases

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.

```
db.execSQL(
      "update tblAMIGO set name = (name || 'XXX') where phone >= '555-1111' ");
```

This statement appends 'XXX' to the name of those whose phone number is equal or greater than '555-1111'.

Note

The symbol **||** is the SQL *concatenate* operator

SQL Databases

SQLite Action Queries Using: ExecSQL cont. 1

Alternatively, the SQL action-statement used in **ExecSQL** could be 'pasted' from pieces as follows:

```
String theValue  =  " …";

db.execSQL( "update tblAMIGO set name = (name || 'XXX') " +
                  " where phone >= '" + theValue + "' " );
```

The same strategy could be applied to other SQL action-statements such as:

"delete from … where…",
"insert into ….values…", etc.

SQL Databases

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)
```

SQL Databases

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.

```
ContentValues myArgs= new ContentValues();

myArgs.put("name", "ABC");
myArgs.put("phone", "555-7777");
```

myArgs

Key	Value
name	ABC
phone	555-7777

SQL Databases

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, *values* could be defined but empty. If the row to be inserted *is empty* (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 (*name, value*) containing the column values for the row that is to be inserted.

SQL Databases

Android's INSERT Operation

```
1 → ContentValues rowValues= new ContentValues();

    rowValues.put("name", "ABC");
    rowValues.put("phone", "555-1010");
2 → long rowPosition = db.insert("tblAMIGO", null, rowValues);

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

4 → rowValues.clear();

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

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

SQL Databases

Android's INSERT Operation

Comments

1. A set of <key, values> called **rowValues** is creted 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 records 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.

SQL Databases

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 (*name, value*) 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.

SQL Databases

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

```
1→  String [] whereArgs = {"2", "7"};

    ContentValues updValues = new ContentValues();

2→  updValues.put("name", "Maria");

3→  int recAffected =  db.update( "tblAMIGO",
                                  updValues,
                                  "recID > ? and recID < ?",
                                  whereArgs );
```

SQL Databases

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 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*".

SQL Databases

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.

SQL Databases

Android's DELETE Operation

Example

Consider the following SQL statement:

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

An equivalent implementation using the Androi'd **delete method** follows:

```
String [] whereArgs = {"2", "7"};

int 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).