SESSION #5

PERSISTENT DATA

OUTLINE

- Shared Preferences
- Internal Storage
- External Storage
- SQLite Database

PERSISTENT DATA

- Android provides several options for you to save persistent application data.
  - **Shared Preferences**: Store private primitive data in key-value pairs.
  - **Internal Storage**: Store private data on the device memory.
  - **External Storage**: Store public data on the shared external storage (SD-Card).
  - **SQLite Databases**: Store structured data in a private database.
  - **Network Services**: Store data on the web with your own network server.
- The solution you choose depends on your specific needs:
  - Whether the data should be private to your application or accessible to other applications (and the user).
  - How much space your data requires.

SHARED PREFERENCES

- The **SharedPreferences** class provides a general framework that allows you to save and retrieve persistent key-value pairs of primitive data types.
  - You can use SharedPreferences to save any primitive data: booleans, floats, ints, longs, and strings.
- This data will persist across user sessions (even if your application is killed).
- The **SharedPreferences** data are not sharable across applications, unless you expose them as a 'content provider'.
- Shared preferences are saved in a xml file inside of the subdirectory "*shared_prefs*" in the application data package directory (/data/data/app_package/shared_prefs/prefs_name.xml)

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="nombre">prueba</string>
    <string name="email">modificado@email.com</string>
</map>
```

SHARED PREFERENCES

ACCESSING SP

- To get a SharedPreferences object for your application, use one of two methods:
  - *public SharedPreferences **getSharedPreferences**(String name, int mode)*

    Use this if you need multiple preferences files identified by name, which you specify with the first parameter.
  - *public SharedPreferences **getPreferences**(int mode)*

    Use this if you need only one preferences file for your Activity. Therefore, you don't need to supply a name.
- The mode parameter sets the following modes:
  - **MODE_PRIVATE** (0): the created file can only be accessed by the calling application (or all applications sharing the same user ID).,
  - **MODE_WORLD_READABLE** (1) and **MODE_WORLD_WRITEABLE** (2) to allow all applications to read or write shared preferences data. Its usage is strongly discouraged.

SHARED PREFERENCES

WRITING SP

- Shared preferences data is accessed in form of key-value pairs.
- To write data do the following steps:
  1. Obtain the SharedPreferences Editor:
     - Call ***edit()*** to get a **SharedPreferences.Editor**.
  2. Add the key-value pairs with the methods such as ***putBoolean***() and ***putString***().
  3. Write the new values to the SharedPreferences file with commit:
     - Commit the new values with commit()

```
SharedPreferences prefs =
     getSharedPreferences("MisPreferencias",Context.MODE_PRIVATE);

SharedPreferences.Editor editor = prefs.edit();
editor.putString("email", "modificado@email.com");
editor.putString("nombre", "Prueba");
editor.commit();
```

SHARED PREFERENCES

READING SP

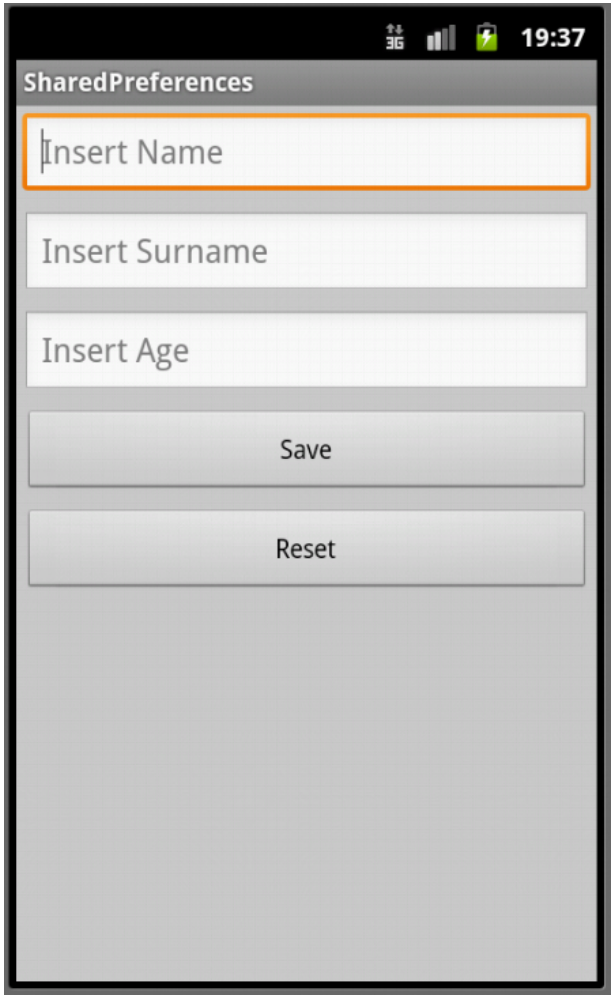
- To read values from the SharedPreferences we only have to use the methods such as **getBoolean()** and **getString()**:
  - *Map<String, ?>* ***getAll()***: Retrieve all values from the preferences.
  - *Boolean* ***getBoolean(String*** *key, boolean defValue)*: Retrieve a boolean value from the preferences.
  - *float* ***getFloat(String*** *key, float defValue)*: Retrieve a float value from the preferences.
  - *int* ***getInt(String*** *key, int defValue)*: Retrieve an int value from the preferences
  - *long* ***getLong(String*** *key, long Value)*: Retrieve a long value from the preferences
  - *String* ***getString(String*** *key, String defValue)*: Retrieve a String value from the preferences

```
SharedPreferences prefs =
     getSharedPreferences("MisPreferencias",Context.MODE_PRIVATE);

String correo = prefs.getString("email", "por_defecto@email.com");
```

SHARED PREFERENCES

- Create an application to illustrated the utilization of SharedPreferences:
  - Stores de user name, surname and age as shared preference
  - When application starts it reads user information from shared preferences.
  - This information can be reset



https://developer.android.com/training/basics/data-storage/shared-preferences.html

SHARED PREFERENCES

- Start a new project and Activity called "*HelloSharedPreferences*"
  1. Open the *res/layout/main.xml* file and replace it with the right code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:orientation="vertical"  android:layout_width="fill_parent"
              android:layout_height="fill_parent"  android:background="#CCC">
   <EditText android:layout_height="wrap_content" android:layout_width="fill_parent"
              android:layout_margin="3dp" android:layout_marginTop="6dp"
              android:hint="Insert Name" android:id="@+id/editTextName"></EditText>
   <EditText android:layout_height="wrap_content" android:layout_width="fill_parent"
              android:layout_margin="3dp" android:hint="Insert Surname" android:id="@
              +id/editTextSurName"></EditText>
   <EditText android:layout_height="wrap_content" android:layout_width="fill_parent"
              android:hint="Insert Age" android:layout_margin="3dp" android:id="@+id/
              editTextAge"></EditText>
   <Button android:id="@+id/button1" android:layout_height="wrap_content"
              android:text="Save" android:layout_width="fill_parent"
              android:layout_margin="3dp" android:onClick="save"></Button>
   <Button android:id="@+id/button2" android:layout_height="wrap_content"
              android:text="Reset" android:layout_width="fill_parent"
              android:layout_margin="3dp" android:onClick="reset"></Button>
</LinearLayout>
```

SHARED PREFERENCES

- When the application starts, access the sharedpreferences object and editor.
  1. Insert the onCreate callback call to init method:

```
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        init();
}
```

  2. In the init method get the SharedPreferences object and editor and read the user information:

```
private void init() {
                // Get SharedPreferences Editor
                Prefs= getSharedPreferences(PREF_NAME, MODE);
                Editor = Prefs.edit();
                // Init View fields
                name = (EditText) findViewById(R.id.editTextName);
                surname = (EditText) findViewById(R.id.editTextSurName);
                age = (EditText) findViewById(R.id.editTextAge);
                age.setInputType(InputType.TYPE_CLASS_NUMBER);
               // Read person info from preferences.
                readPerson();
}
```

SHARED PREFERENCES

- Insert the methods for read users information from the preferences:

```
/*Read the data refer to saved person and visualize them into Edittexts */
private void readPerson() {
        name.setText(Prefs.getString(NAME, null));
        surname.setText(Prefs.getString(SURNAME, null));
        String agePref = ""     + Prefs.getInt(AGE, 0);
        age.setText((agePref.equals("0")) ? null : agePref);
}
```

  - We also need to define the constants for all fields name and preferences file.

```
public static final String PREF_NAME = "PEOPLE_PREFERENCES";
public static final int MODE = Context.MODE_PRIVATE;

public static final String NAME = "NAME";
public static final String SURNAME = "SURNAME";
public static final String AGE = "AGE";
```

SHARED PREFERENCES

- Finally, define the method for save and reset the sharedpreferences fields:

```
public void save(View view) {

        String nameText = name.getText().toString();
        String surnameText = surname.getText().toString();
        String ageText = age.getText().toString();

        if (nameText != null)  Editor.putString(NAME, nameText).commit();
        if (surnameText != null)  Editor.putString(SURNAME, surnameText).commit();
        if (ageText != null && !ageText.equals(""))
                Editor.putInt(AGE, Integer.parseInt(ageText)).commit();
}

public void reset(View view) {
        /* A better way to delete all is:  Editor.clear().commit();  */
        Editor.remove(NAME).commit();
        Editor.remove(SURNAME).commit();
        Editor.remove(AGE).commit();
        readPerson();
}
```

INTERNAL STORAGE

- Android applications can save files directly on the device's internal storage.
  - By default, files saved to the internal storage are private to your application and other applications cannot access them (nor can the user).
  - Internal files are saved inside the application data directory:
    - /data/data/{***your-app-path***}/files/{***file-name***}
  - These internal files are deleted when the user uninstalls the application.
- The internal file storage API is provided by the Context class, which basically provides access to our application environment.

INTERNAL STORAGE

OPEN/CREATING FILES

- To open/create a file in the internal storage, we have to use the ***openFileInput***() and ***openFileOutput***() methods from context object:

  public FileInputStream **openFileInput** (String name)

  public FileOutputStream **openFileOutput** (String name, int mode)

  - Opens an input or output file. If the file does not exist the openFileOutput creates it. Returns a file stream for reading or writing in the file.
    - **Context.MODE_PRIVATE** means our internal files are only visible to us.
    - **Context.MODE_APPEND** we use it when we need to add data to the end of the file, otherwise we will overwrite the file every time we write to it.
    - We can also use **MODE_WORLD_READABLE** and **MODE_WORLD_WRITABLE**.

```
String FILENAME = "hello_file.txt";

// saving into file
FileOutputStream fos = context.openFileOutput(FILENAME, Context.MODE_PRIVATE);

// write I/O using write() and close()

// retrieving from file
FileInputStream fis = context.openFileInput(FILENAME);

// read I/O using read() and close()
```

INTERNAL STORAGE

READING & WRITING FILES

- File manipulation in Android is based on standard Java Input/ Output API (I/O Streams).

```
try {
      // OPEN FILE INPUT STREAM THIS TIME
      FileInputStream fis = openFileInput(fileName);
      InputStreamReader isr = new InputStreamReader(fis);

      // READ STRING OF UNKNOWN LENGTH
      StringBuilder sb = new StringBuilder();
      char[] inputBuffer = new char[2048];
      int l;
      // FILL BUFFER WITH DATA
      while ((l = isr.read(inputBuffer)) != -1) {
            sb.append(inputBuffer, 0, l);
      }

      // CONVERT BYTES TO STRING
      Log.i("LOG_TAG", "Read string: " + sb.toString());
      fis.close();
}
catch (IOException e) {
      e.printStackTrace();
}
```

INTERNAL STORAGE

OTHER METHODS

- File ***getFilesDir***()
  - Gets the absolute path to the file system directory where your internal files are saved.
- File ***getDir***(String name, int mode)
  - Creates (or opens an existing) directory within your internal storage space
- boolean ***deleteFile***(String name)
  - Deletes a file saved on the internal storage.
- String[] ***fileList***()
  - Returns an array of files currently saved by your application.

INTERNAL STORAGE

CACHE & RAW FILES

- Saving cache files
  - To cache some data, rather than store it persistently, you should use ***getCacheDir()*** to open a File that represents the internal directory where your application should save temporary cache files.
  - When the device is low on internal storage space, Android may delete these cache files to recover space.
- Accessing raw files
  - The ***AssetManager*** provides access to an application's raw asset files.
  - This class presents a lower-level API that allows you to open and read raw files that have been bundled with the application as a simple stream of bytes.

```
AssetManager assetManager = getAssets();

// To load image
// get input stream
InputStream ims = assetManager.open("android_logo_small.jpg");

// create drawable from stream
Drawable d = Drawable.createFromStream(ims, null);

// set the drawable to imageview
imgAssets.setImageDrawable(d);
```

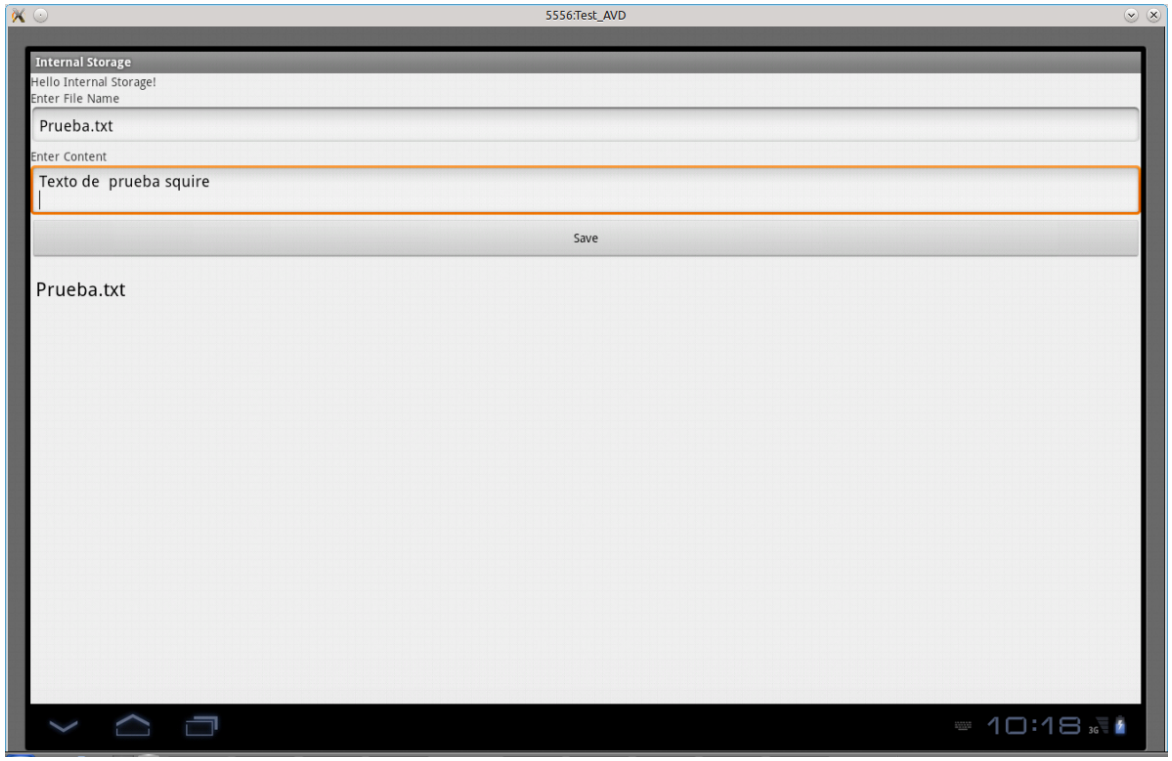
INTERNAL STORAGE

INTERNAL STORAGE VS SHARED PREFERENCES

- ✧ **SharedPreferences**: Simple and less overhead
  - In a shared preference, there is much less overhead and so reading/writing to a simple Map object is much more efficient than reading/writing to a disk.
  - However, because you are limited to simple primitive values, you are essentially trading flexibility for efficiency.
- ✧ **Internal Storage**: More efficient and flexible for larger data.
  - With internal and external storage mechanisms, you can save not only much bigger chunks of data (that is, entire XML files) but also much more complicated forms of data (that is, media files, image files, and so on).

INTERNAL STORAGE

✧ Creation of an application to illustrated the utilization:



EXTERNAL STORAGE

- Every Android-compatible device supports a shared "external storage" that you can use to save files.
  - This can be a removable storage media (such as an SD card) or an internal (non-removable) storage.
  - Files saved to the external storage are world-readable and can be modified by the user when they enable USB mass storage to transfer files on a computer.

- ***Caution***: External storage can become unavailable if the user mounts the external storage on a computer or removes the media.
  - You have to check the availability of the external files / storage
  - There is no security enforced upon files you save to the external storage.
  - All applications can read and write files placed on the external storage and the user can remove them.

EXTERNAL STORAGE

CHECKING MEDIA AVAILABILITY

- Before you do any work with the external storage, you should always call ***getExternalStorageState()*** to check whether the media is available.
  - The media might be mounted to a computer, missing, read-only, or in some other state. The ***getExternalStorageState()*** method returns the media state:
    - **MEDIA_MOUNTED**: The media is present and mounted at its mount point with read/write access.
    - **MEDIA_UNMOUNTED**: The media is present but not mounted.
    - **MEDIA_REMOVED**: The media is not present.
    - **MEDIA_SHARED**: The media is being shared via USB mass storage (connected to a computer), it is not mounted.
    - **MEDIA_MOUNTED_READ_ONLY**: The media is present and mounted with read only access.

```
boolean mExternalStorageAvailable = false;
boolean mExternalStorageWriteable = false;
String state = Environment.getExternalStorageState();

if (Environment.MEDIA_MOUNTED.equals(state)) {
    // We can read and write the media
    mExternalStorageAvailable = mExternalStorageWriteable = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
    // We can only read the media
    mExternalStorageAvailable = true;
    mExternalStorageWriteable = false;
} else {
    // Something else is wrong. It may be one of many other states, but all we need to know is we can neither read nor write
    mExternalStorageAvailable = mExternalStorageWriteable = false;
}
```

EXTERNAL STORAGE

ACCESSING FILES

- ✧ If API Level 8 or greater:
  - Use ***getExternalFilesDir***() to open a *File* that represents the external storage directory where you should save your files.

    public static File **getExternalFilesDir** (String type)
    - The type paremeter specifies the type of subdirectory you want, such as **DIRECTORY_MUSIC** and **DIRECTORY_RINGTONES**.
      - Pass null to receive the root of your application's file directory.
    - This method will create the appropriate directory if necessary.
      - By specifying the type of directory, you ensure that the Android's media scanner will properly categorize your files in the system.
    - If user uninstalls application, this directory and all its contents will be deleted.
- ✧ If API Level 7 or lower:
  - Use ***getExternalStorageDirectory***(), to open a *File* representing the root of the external storage.

    public static File **getExternalStorageDirectory()**
    - You should then write your data in the following directory: **/Android/data/<package_name>/files/**

EXTERNAL STORAGE

```
void createExternalStoragePrivateFile() {
    // Create a path where we will place our private file on external storage.
    File file = new File(getExternalFilesDir(null), "DemoFile.jpg");

    try {
        // Very simple code to copy a picture from the application's resource into the external file.
        InputStream is = getResources().openRawResource(R.drawable.balloons);
        OutputStream os = new FileOutputStream(file);
        byte[] data = new byte[is.available()];
        is.read(data);
        os.write(data);
        is.close();
        os.close();
    } catch (IOException e) {
        // Unable to create file, likely because external storage is  not currently mounted.
        Log.w("ExternalStorage", "Error writing " + file, e);
    }
}

void deleteExternalStoragePrivateFile() {
    // Get path for the file on external storage.  If external  storage is not currently mounted this will fail.
    File file = new File(getExternalFilesDir(null), "DemoFile.jpg");
    if (file != null) {
        file.delete();
    }
}
```

EXTERNAL STORAGE

```
boolean hasExternalStoragePrivateFile() {
    // Get path for the file on external storage.  If external
    // storage is not currently mounted this will fail.
    File file = new File(getExternalFilesDir(null), "DemoFile.jpg");
    if (file != null) {
        return file.exists();
    }
    return false;
}
```

EXTERNAL STORAGE

SAVING SHARED FILES

- To save files that are not specific to your application and that should not be deleted when your application is uninstalled, save them to one of the **public directories** on the external storage.
  - These directories lay at the root of the external storage.
- If API Level 8 or greater:
  - Use ***getExternalStoragePublicDirectory*** (), passing it the type of public directory you want, such as **DIRECTORY_MUSIC**, **DIRECTORY_PICTURES**, **DIRECTORY_RINGTONES**, or others.
    - public static File **getExternalStoragePublicDirectory** (String type)
- If API Level 7 or lower:
  - Use ***getExternalStorageDirectory***(), to open a *File* representing the root of the external storage, then save your shared files in one of the following directories: **Music/, Podcasts/, Ringtones/, Alarms/, Notifications/, Pictures/, Movies/** and **Download/**

EXTERNAL STORAGE

EXTERNAL VS INTERNAL STORAGE

- **Storage Space** (memory).
  - The amount of internal memory can often be quite low.
  - External storage, depends solely on what SD card the user has in their phone. Typically, if an SD card is present, then the amount of external storage can be many times greater than the amount of internal storage.
- **Storage Access speed.**
  - The access speeds are highly dependent on the type of internal flash memory as well as the classification of the SD card for external storage.
- **Storage Accessibility.**
  - The internal storage data is only accessible by your application
    - This is extremely safe from potentially malicious external applications.
    - But, if the application is uninstalled, then that internal memory is deleted as well.
  - For external storage, the visibility is inherently world readable and writeable, and so any files saved are exposed both to external applications as well as to the user.
    - There is no guarantee that your files will remain safe and uncorrupted.

EXTERNAL STORAGE

PERMISSIONS

- To get access to external storage in your android-device you need to add a uses permission into the **AndroidManifest.xml** file for your app.
- The permission you need to add is:
  - **READ_EXTERNAL_STORAGE**

    Allows an application to read from external storage.
  - **WRITE_EXTERNAL_STORAGE**

    Allows an application to write to external storage. An application with write permission also have read permission.

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

EXTERNAL STORAGE

- Create an application to get access to external storage (SD-Card):
  - Check if the external media are available
  - Writes a file to the sd-card
  - Reads a raw file
  - Finally, list files in the external storage directory.
- For check if the file is created you can connect to the android emulator:

  > adb shell

  > cd /mnt/sdcard/download/

  > cat myData.txt

  (*) You need android platform tools path in your PATH

ANDROID DB SUPPORT

- Android provides full support for ***SQLite*** databases.
  - SQLite supports standard relational database features like SQL syntax, transactions and prepared statements.
  - It requires only little memory at runtime (approx. 250 KByte).
  - Any databases you create will be accessible by name to any class in the application, but not outside the application.
- SQLite in Android
  - SQLite is available on every Android device.
  - Using an SQLite does not require any database setup or administration
    - You only have to define the SQL statements for creating and updating the database. Afterwards the database is automatically managed for you by the Android platform.
  - Access to an SQLite database involves accessing the filesystem.
    - This can be slow. Therefore it is recommended to perform database operations asynchronously, for example inside the *AsyncTask* class.
    - If your application creates a database, this database is by default saved in the directory **DATA/data/APP_NAME/databases/FILENAME**.

ARCHITECTURE

- The package **android.database** contains all general classes for working with databases.
- The **android.database.sqlite** contains the SQLite specific classes:
  - **SQLiteOpenHelper:** A helper class to manage database creation and version management.
  - **SQLiteDatabase:** Class that exposes methods to manage a SQLite database.
  - **rawQuery()** and **Query():** Methods to query the given table, returning a Cursor over the result set.
  - **Cursor:** Interface that provides random read-write access to the result set returned by a database query.
  - **SQLiteQueryBuilder:** A convenience class that helps build SQL queries to be sent to SQLiteDatabase objects.
  - **SimpleCursorAdapter:** An easy adapter to map columns from a cursor to TextViews or ImageViews defined in an XML file.

SQLITE OPEN HELPER

- The recommended method to create a new SQLite database is to create a subclass of **SQLiteOpenHelper** and override the ***onCreate***() method, in which you can execute a **SQLite** command to create tables in the database.
  - You can then get an instance of your **SQLiteOpenHelper** implementation using the constructor you've defined.
    - In the constructor of your subclass you call the super() method of **SQLiteOpenHelper**, specifying the database name and the current database version.
  - In this class you need to override the ***onCreate***() and ***onUpgrade***() methods.
    - ***onCreate***() is called by the framework, if the database does not exists.
    - ***onUpgrade***() is called, if the database version is increased in your application code. This method allows you to update the database schema.
  - To write to and read from the database, call ***getWritableDatabase***() and ***getReadableDatabase***(), respectively.
    - These both return a **SQLiteDatabase** object that represents the database and provides methods for SQLite operations.

SQLITE OPEN HELPER

- The database tables should use the identifier _id for the primary key of the table. Several Android functions rely on this standard.
  - We do recommend including an autoincrement value key field that can be used as a unique ID to quickly find a record.
- It is best practice to create a separate class per table.

```
public class DictionaryOpenHelper extends SQLiteOpenHelper {

   private static final int DATABASE_VERSION = 2;
   private static final String DICTIONARY_TABLE_NAME = "dictionary";
   private static final String KEY_ID = "_id";
   private static final String KEY_WORD = "word";
   private static final String KEY_DEFINITION = "definition";
   private static final String DICTIONARY_TABLE_CREATE =
            "CREATE TABLE " + DICTIONARY_TABLE_NAME + " (" +
            ID + " integer primary key autoincrement, " +
            KEY_WORD + " TEXT, " +
            KEY_DEFINITION + " TEXT);";

   DictionaryOpenHelper(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);
   }

   @Override
   public void onCreate(SQLiteDatabase db) {
      db.execSQL(DICTIONARY_TABLE_CREATE);
   }
}
```

SQLITE DB (I)

- SQLiteDatabase is the base class for working with a SQLite database in Android and provides methods to open, query, update and close the database.

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

  - Method for inserting a row into the database.
    - Table: the table to insert the row into
    - Values: this map contains the initial column values for the row. The keys should be the column names and the values the column values
  - The object ContentValues allows to define key/values.
    - The "key" represents the table column identifier and the "value" represents the content for the table record in this column.

```
public long NewDefinition(string word, string definition)
      final ContentValues values = new ContentValues();
      values.put(KEY_WORD, word);
      values.put(KEY_DEFINITION, definition);
      return db.insert(DICTIONARY_TABLE_NAME,  null, values);
}
```

SQLITE DB (II)

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

- Method for updating rows in the database.
  - whereClause the optional WHERE clause to apply when updating. Passing null will update all rows.

```
public void UpdateDefinition(int Id, String word, String definition) {
      final ContentValues values = new ContentValues();
      values.put(KEY_WORD, word);
      values.put(KEY_DEFINITION, definition);
      db.update(DICTIONARY_TABLE_NAME, values,  KEY_ID + " = ?" , new String[]
                  {String.valueOf(Id)});
}
```

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

- Method for deleting rows in the database.

```
public void DeleteDefinition(int Id) {
      if (Id()>0)
      {
            db.delete(DICTIONARY_TABLE_NAME,  KEY_ID + " = ?" , new   String[]
                        {String.valueOf(Id)});
      }
}
```

SQLITE DB (III)

- In addition it provides the ***execSQL***() method, which allows to execute an SQL statement directly.

  public void **execSQL** (String sql)

  - Execute a single SQL statement that is NOT a SELECT or any other SQL statement that returns data.
    - Sql: the SQL statement to be executed. Multiple statements separated by semicolons are not supported.

```
public static void onCreate(SQLiteDatabase db){
StringBuilder sb = new StringBuilder();
sb.append("CREATE TABLE " + DICTIONARY_TABLE_NAME  + " (");
sb.append(ID + " integer primary key autoincrement, " );
sb.append(KEY_WORD + " TEXT NOT NULL, " );
sb.append(KEY_DEFINITION + " TEXT");
sb.append(");");
db.execSQL(sb.toString());
}

public static void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion){
db.execSQL("DROP TABLE IF EXISTS " +DICTIONARY_TABLE_NAME);
this.onCreate(db);
}
```

SQLITE DB (IV)

- The SQLiteDatabase object also allows manage transactions
  - void **beginTransaction**(), public void **beginTransactionNonExclusive** ()

    Begins a transaction in EXCLUSIVE or INMEDIATE mode.
  - void **setTransactionSuccessful**()

    Marks the current transaction As successful.
  - void **endTransaction**()

    End a transaction.

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

DB QUERY (I)

- Queries can be created via the ***rawQuery***() and ***query***() methods or via the SQLiteQueryBuilder class .

  public Cursor **rawQuery**(String sql, String[] selectionArgs)

  - Directly accepts an SQL select statement as input. Runs the provided SQL and returns a Cursor over the result set.
    - sql: the SQL query. The SQL string must not be ; terminated
    - selectionArgs: You may include "?" in where clause in the query, which will be replaced by the values from selectionArgs. The values will be bound as Strings.

```
public Contact getDefinition(int id) {
   SQLiteDatabase db = this.getReadableDatabase();

   Cursor cursor = getReadableDatabase().rawQuery("select * from dictionary where
                   _id = ?", new String[] { id });
   if (cursor != null)
      cursor.moveToFirst();

   Definition def = new Definition(Integer.parseInt(cursor.getString(0)),
         cursor.getString(1), cursor.getString(2));
   // return definition
   return def;
}
```

DB QUERY (II)

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

– Query the given table, returning a Cursor over the result set.

PARAMETER	DESCRIPTION
Boolean distinct	True if you want each row to be unique, false otherwise.
String table	The table name to compile the query against.
String[] columns	A list of which table columns to return. Passing "null" will return all columns.
String selection	A filter declaring which rows to return formatted as an SQL WHERE clause (excluding the WHERE itself). Passing null will return all rows.
String[] selectionArgs	You may include ?s in the "whereClause"". These placeholders will get replaced by the values from the selectionArgs array.
String groupBy	A filter declaring how to group rows, null will cause no grouping.
String having	Filter for the groups, null means no filter.
String orderBy	Table columns which will be used to order the data, null means no ordering.
String limit	Limits the number of rows returned by the query. Null denotes no LIMIT.

```
public Contact getDefinition(int id) {
   SQLiteDatabase db = this.getReadableDatabase();

   Cursor cursor = db.query(DICTIONARY_TABLE_NAME, new String[] { KEY_ID, KEY_WORD,
     KEY_DEFINITION }, KEY_ID + "=?", new String[] { String.valueOf(id) }, null, null, null, null);
    .........
}
```

SQLITE QUERY BUILDER

- The ***SQLiteQueryBuilder*** offers much richer query-building options, particularly for queries involving the union of multiple sub-query results.
  - The ***SQLiteQueryBuilder*** interface fits well with the ***ContentProvider*** interface for executing queries

```
SQLiteQueryBuilder sqLiteQueryBuilder = new SQLiteQueryBuilder();
sqLiteQueryBuilder.setTables("A, B");
sqLiteQueryBuilder.setDistinct(true);

String selection =  "A.ID = B.ID";

Cursor c = sqLiteQueryBuilder.query(db,
    new String[]{"A.x"},
    selection,
    null, null,  null,  null);
```

Query: SELECT DISTINCT A.x FROM A,B WHERE A.ID = B.ID

SQLITE QUERY BUILDER

✧ Create a query for joining two tables:

```
//Create new querybuilder
SQLiteQueryBuilder _QB = new SQLiteQueryBuilder();

//Specify books table and add join to categories table (use full_id for joining
categories table)
_QB.setTables(BookColumns.TABLENAME +
        " LEFT OUTER JOIN " + CategoryColumns.TABLENAME + " ON " +
        BookColumns.CATEGORY + " = " + CategoryColumns.FULL_ID);

//Order by records by title
_OrderBy = BookColumns.BOOK_TITLE + " ASC";

//Open database connection
SQLiteDatabase _DB = fDatabaseHelper.getReadableDatabase();

//Get cursor
Cursor _Result = _QB.query(_DB, null, null, null, null, null, _OrderBy);

http://blog.cubeactive.com/android-creating-a-join-with-sqlite/
```

CURSORS

- A query returns a ***Cursor*** object.
  - A ***Cursor*** represents the result of a query and basically points to one row of the query result.
  - This way Android can buffer the query results efficiently; as it does not have to load all data into memory.
- Functionality:
  - To get the number of elements of the resulting query use the ***getCount***() method.
  - To move between individual data rows, you can use the ***moveToFirst***() and ***moveToNext***() methods. The ***isAfterLast***() method allows to check if the end of the query result has been reached.
  - ***Cursor*** provides typed ***get****() methods, e.g. ***getLong***(columnIndex), ***getString***(columnIndex) to access the column data for the current position of the result. The "*columnIndex*" is the number of the column you are accessing.
  - Cursor also provides the ***getColumnIndexOrThrow***(String) method which allows to get the column index for a column name of the table.
  - A Cursor needs to be closed with the ***close***() method call.

SIMPLE CURSOR ADAPTER

- To work with databases and ListViews you can use the ***SimpleCursorAdapter***. The ***SimpleCursorAdapter*** allows to set a layout for each row of the ListViews.
  - ListViews are Views which allow to display a list of elements.
  - ListActivities are specialized Activities which make the usage of ListViews easier.
- You also define an array which contains the column names and another array which contains the IDs of Views which should be filled with the data.
- The SimpleCursorAdapter class will map the columns to the Views based on the Cursor passed to it.
  - To obtain the Cursor you should use the Loader class.
- ***SimpleCursorAdapter***(Context context, int layout, Cursor c, String[] from, int[] to)
  - **Context** – A reference to the containing Activity.
  - **Layout** – The resource ID of the row view to use.
  - **Cursor** – A cursor containing the SQLite query for the data to display.
  - **From string array** – An array of strings corresponding to the names of columns in the cursor.
  - **To integer array** – An array of layout IDs that correspond to the controls in the row layout.

This constructor was deprecated in API level 11. This option is discouraged, as it results in Cursor queries being performed on the application's UI thread and thus can cause poor responsiveness or even Application Not Responding errors. As an alternative, use LoaderManager with a CursorLoader.

EXAMPLE: SIMPLE CURSOR ADAPTER

```
public class ListA extends ListActivity {

	@Override
	public void onCreate(Bundle savedInstance) {
		setContentView(R.layout.list_example);

		Cursor cursor = getContentResolver().query(People.CONTENT_URI,
	  new String[] {People._ID, People.NAME, People.NUMBER}, null, null,
	  null);
		startManagingCursor(cursor);

		// the desired columns to be bound
		String[] columns = new String[] { People.NAME, People.NUMBER };

		// the XML defined views which the data will be bound to
		int[] to = new int[] { R.id.name_entry, R.id.number_entry };

		// create the adapter using the cursor pointing to the desired data as
well as the layout information
		SimpleCursorAdapter mAdapter = new SimpleCursorAdapter(this,
	  R.layout.list_example_entry, cursor, columns, to);

		// set this adapter as your ListActivity's adapter
		this.setListAdapter(mAdapter);
	}
}
```

EXAMPLE: HELLO DATABASE

- Creation of an application to illustrated the utilization of Database:

http://www.vogella.com/articles/AndroidSQLite/article.html
https://developer.android.com/training/basics/data-storage/databases.html

EXAMPLE: BEAN

SQLITE DB

```
package com.example.jgervas.datamanagement.database;

public class Comment {
          private long id;
          private String comment;

          public long getId() {
            return id;
          }

          public void setId(long id) {
            this.id = id;
          }

          public String getComment() {
            return comment;
          }

          public void setComment(String comment) {
            this.comment = comment;
          }

          // Used by the ArrayAdapter in the ListView
          @Override
          public String toString() {
            return comment;
          }
        }
```

EXAMPLE: DB HELPER

SQLITE DB

```
package com.example.jgervas.datamanagement.database;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;

public class MySQLiteHelper extends SQLiteOpenHelper {

          public static final String TABLE_COMMENTS = "comments";
          public static final String COLUMN_ID = "_id";
          public static final String COLUMN_COMMENT = "comment";

          private static final String DATABASE_NAME = "commments.db";
          private static final int DATABASE_VERSION = 1;

          // Database creation sql statement
          private static final String DATABASE_CREATE = "create table "
              + TABLE_COMMENTS + "(" + COLUMN_ID
              + " integer primary key autoincrement, " + COLUMN_COMMENT
              + " text not null);";

          public MySQLiteHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
          }

          @Override
          public void onCreate(SQLiteDatabase database) {
            database.execSQL(DATABASE_CREATE);
          }

          @Override
          public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
            Log.w(MySQLiteHelper.class.getName(),
                "Upgrading database from version " + oldVersion + " to "
                    + newVersion + ", which will destroy all old data");
            db.execSQL("DROP TABLE IF EXISTS " + TABLE_COMMENTS);
            onCreate(db);
          }

        }
```

EXAMPLE: DATA SOURCE

SQLITE DB

```
package com.example.jgervas.datamanagement.database;

import java.util.ArrayList;
import java.util.List;
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;

public class CommentsDataSource {

          // Database fields
          private SQLiteDatabase database;
          private MySQLiteHelper dbHelper;
          private String[] allColumns = { MySQLiteHelper.COLUMN_ID,
              MySQLiteHelper.COLUMN_COMMENT };

          public CommentsDataSource(Context context) {
            dbHelper = new MySQLiteHelper(context);
          }

          public void open() throws SQLException {
            database = dbHelper.getWritableDatabase();
          }

          public void close() {
            dbHelper.close();
          }

          public Comment createComment(String comment) {
            ContentValues values = new ContentValues();
            values.put(MySQLiteHelper.COLUMN_COMMENT, comment);
            long insertId = database.insert(MySQLiteHelper.TABLE_COMMENTS, null,
                values);
            Cursor cursor = database.query(MySQLiteHelper.TABLE_COMMENTS,
                allColumns, MySQLiteHelper.COLUMN_ID + " = " + insertId, null,
                null, null, null);
            cursor.moveToFirst();
            Comment newComment = cursorToComment(cursor);
            cursor.close();
            return newComment;
          }
```

EXAMPLE: DATA SOURCE

SQLITE DB

```
  public void deleteComment(Comment comment) {
    long id = comment.getId();
    System.out.println("Comment deleted with id: " + id);
    database.delete(MySQLiteHelper.TABLE_COMMENTS, MySQLiteHelper.COLUMN_ID
        + " = " + id, null);
  }

  public List<Comment> getAllComments() {
    List<Comment> comments = new ArrayList<Comment>();

    Cursor cursor = database.query(MySQLiteHelper.TABLE_COMMENTS,
        allColumns, null, null, null, null, null);

    cursor.moveToFirst();
    while (!cursor.isAfterLast()) {
      Comment comment = cursorToComment(cursor);
      comments.add(comment);
      cursor.moveToNext();
    }
    // Make sure to close the cursor
    cursor.close();
    return comments;
  }

  private Comment cursorToComment(Cursor cursor) {
    Comment comment = new Comment();
    comment.setId(cursor.getLong(0));
    comment.setComment(cursor.getString(1));
    return comment;
  }
}
```

EXAMPLE: LAYOUT

SQLITE DB

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_sqlite"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context="com.example.jgervas.datamanagement.SqliteActivity">

    <LinearLayout
        android:id="@+id/group"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >

        <Button
            android:id="@+id/add"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Add New"
            android:onClick="onClick"/>

        <Button
            android:id="@+id/delete"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Delete First"
            android:onClick="onClick"/>

    </LinearLayout>

    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/helloSqlite" />

</LinearLayout>
```

EXAMPLE: MAIN

SQLITE DB

```
public class SqliteActivity extends ListActivity {

    private CommentsDataSource datasource;

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

        datasource = new CommentsDataSource(this);
        datasource.open();

        List<Comment> values = datasource.getAllComments();

        // Use the ArrayAdapter to show the
        // elements in a ListView
        ArrayAdapter<Comment> adapter = new ArrayAdapter<Comment>(this,
                android.R.layout.simple_list_item_1, values);
        setListAdapter(adapter);
    }

    // Will be called via the onClick attribute
    // of the buttons in xml layout
    public void onClick(View view) {

        ArrayAdapter<Comment> adapter = (ArrayAdapter<Comment>) getListAdapter();
        Comment comment = null;
        switch (view.getId()) {
            case R.id.add:
                String[] comments = new String[] { "Cool", "Very nice", "Hate it" };
                int nextInt = new Random().nextInt(3);
                // Save the new comment to the database
                comment = datasource.createComment(comments[nextInt]);
                adapter.add(comment);
                break;
```

EXAMPLE: MAIN

SQLITE DB

```
case R.id.delete:
                if (getListAdapter().getCount() > 0) {
                    comment = (Comment) getListAdapter().getItem(0);
                    datasource.deleteComment(comment);
                    adapter.remove(comment);
                }
                break;
        }
        adapter.notifyDataSetChanged();
    }

    @Override
    protected void onResume() {
        datasource.open();
        super.onResume();
    }

    @Override
    protected void onPause() {
        datasource.close();
        super.onPause();
    }

}
```