|  |
| --- |
| Android Tutorial – Part 4 |

|  |
| --- |
| 6-17-2018 |



Table of Contents

[Introduction 2](#_Toc517127740)

[Saving Data in Android App 2](#_Toc517127741)

[Store data locally in an Android app 3](#_Toc517127742)

[Different data storage techniques available to Android 3](#_Toc517127743)

[SQLite database 5](#_Toc517127744)

[SQLite’s noticeable features: 5](#_Toc517127745)

[SQLite distinctive features 6](#_Toc517127746)

[Android Database storage with SQLite 6](#_Toc517127747)

[Important classes used in Android SQLite 6](#_Toc517127748)

[Saving messages sent in Database 7](#_Toc517127749)

[The database table structure to save a message 7](#_Toc517127750)

[Model Class for a Message 8](#_Toc517127751)

[Mapping between relational DB table and java model class 12](#_Toc517127752)

[SQLite Helper Class - MyDatabaseHelper 12](#_Toc517127753)

[DB operation helper class – MyDataBaseOperations 18](#_Toc517127754)

[Call createMessage() method from “SendMessageActivity” class 26](#_Toc517127755)

[Retrieving saved messages from Database 28](#_Toc517127756)

[Design parent ListView 28](#_Toc517127757)

[Design Single Row Item 29](#_Toc517127758)

[Write a Custom Array Adapter 32](#_Toc517127759)

[Initializing List with messages from DB 36](#_Toc517127760)

[References 39](#_Toc517127761)

# Introduction

This is the part four of the android tutorial series. It is a continuation from last week. In order to follow this successfully, it is required to have,

* A basic understanding given about android in last session.
* The environment set up.
* The project created during last tutorial, opened in Android Studio.
* AVD or an Actual device ready for app deployment.

To catch up, in the last session (Android Tutorial Part 3),

* AutoCompleteTextView
  + Attributes
  + Methods
* Adapters in Android
* App Permissions in Andorid
* Set a random string list as data source for AutoCompleteTextView
* Adding special permissions to “**AndroidManifest.xml**”
* Checking for app permissions
* Requesting permission from user
* Reading contact list on device
* Send an SMS from app
* Source code for this part can be found in Git Repository given below :- <https://github.com/nadee158/android_tutorial_part_3.git>

With that knowledge in hand, in this session below areas will be covered,

* Different storage techniques in Android
* SQLite database
* Important classes and their methods of Android’s SQLite package
* Table structure to save a message
* Model class structure to save a message
* SQLiteOpenHelper class, and its implementation
* DB Operation helper class
* Android ListView
* Custom Adapter

# Saving Data in Android App

In the previous parts of the tutorial, we have created a UI to type in/select a contact number, and type a message and send it. After sending, we directed the user to a new screen, in which the sent message and contact number got displayed.

When you refer to the real app available in your phone to send SMS, you must have noticed that you are able to see all the messages sent so far. Let’s implement that feature to our app as well. In this tutorial lets save all the messages we sent, and after sending a message, let’s display all the messages sent so far (through our app) sorted by most recently sent on top.

For that, a data storing and retrieving mechanism should be used.

## Store data locally in an Android app

Almost every android application will have to store data in one way or another. This data can be of different forms depending on the app like,

* + user settings
  + application settings
  + user data
  + images
  + cache of data fetched over a remote service

Some apps might generate data that ultimately belongs to the user, and so, would prefer to store the data (perhaps documents or media) in a public place that the user can access at any time, using other apps.

Other apps might want to store data, but do not want this data to be read by other apps (or even the user).

Android platform provides several options for the developer to save app data, with each method having its advantages and disadvantages.

The solution you choose depends on your specific needs, such as;

* Space :- how much space your data requires
* Type :- what kind of data you need to store
* Security: - whether the data should be private to your app or accessible to other apps and the user.

### Different data storage techniques available to Android

There are mainly four different ways to store data in an Android app as listed below;

* **Shared preferences**: - Store private primitive data in key-value pairs.
* **Internal file storage**: - Store app-private files on the device file system.
* **External file storage**: - Store files on the shared external file system. This is usually for shared user files, such as photos.
* **Databases**: - Store structured data in a private database.

#### Shared preferences

If there’s **no need to store a lot of data** and it **doesn't require structure**, then **SharedPreferences** can be used. The SharedPreferences APIs allows to read and write persistent **key-value pairs of primitive data types** such as;

* boolean
* float
* int
* long
* string
* The key-value pairs are written to XML files that persist across user sessions, even if the app is killed.
* Developer can manually specify a name for the file or use per-activity files to save data.

The "shared preferences" API is not strictly for saving "user preferences, “but can be used to save any kind of simple data.

If you do want to save user preferences for your app, then you should read how to create a settings UI, which uses **PreferenceActivity** to build a settings screen and automatically persist the user's settings.

#### Internal file storage

By default, files saved to the internal storage are private to a app, and other apps cannot access them (nor can the user, unless they have root access).

* The system provides a private directory on the file system for each app.
* Internal storage a good place for internal app data that the user doesn't need to directly access.
* When the user **uninstalls the app**, the files saved on the i**nternal** **storage** are **removed**.
* Therefore, internal storage should not be used to save anything the user expects to persist independently of the app.

**Data stored using the Internal Storage method is completely private to the application, and are deleted from the device when the app is uninstalled.**

##### Internal cache files

This is used to keep some data temporarily, rather than store it persistently. A special cache directory is used to save the data.

* Each app has a private cache directory specifically for these kinds of files.
* When the device is low on internal storage space, Android may delete these cache files to recover space.
* Always cache files should be maintained by the app, and should stay within a reasonable limit of space consumed, such as 1MB.
* When the user uninstalls the app, these files are removed.

#### External file storage

Every Android device supports a shared "external storage" space that can be used to save files.

* Called external because **it's not a guaranteed to be accessible**
  + E.g.:- like astorage space that users can mount to a computer as an external storage device
* Can be physically removable (such as an SD card).
* After saved to the external storage, such files are readable by anyone
* Files can be modified by the user when they enable USB mass storage to transfer files on a computer.

Therefore, before using external storage, should check for the availability of the external storage directories and accessed files always.

Use external storage

* For user data that should be accessible to other apps.
* Files that should be saved even if the user uninstalls the app,
  + e.g.:- captured photos or downloaded files.
  + The system provides standard public directories for these kinds of files, so the user has one location for all their photos, ringtones, music, and such.
* It is possible to save files to the external storage in an app-specific directory that the system deletes when the user uninstalls the app.
* It is a useful alternative to internal storage if more space is required
* To save (and/or read) files to the device’s external storage, the app must request for the **WRITE\_EXTERNAL\_STORAGE** permission.
* If requirement is only to **read** from the External Storage without writing, request for the **READ\_EXTERNAL\_STORAGE** permission.
* The **WRITE\_EXTERNAL\_STORAGE** permission grants both **read/write** access.

#### Databases

Android provides full support for SQLite databases. Any database create by an app is accessible only by that app itself.

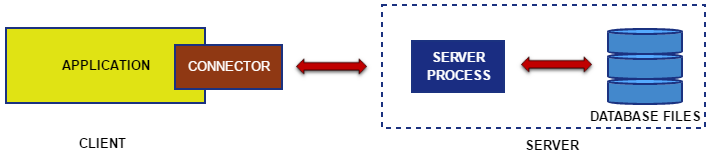
* Databases created are app specific, and are available to any class within the app, but not to outside applications.
* To use a SQLite database for data storage in app, some SQL knowledge is required.

## SQLite database

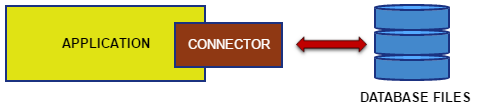
SQLite is a software library that provides a **relational database management system**. The **lite** in SQ**Lite** means **light weight** in terms of **setup**, **database administration**, and **required resource**.

### SQLite’s noticeable features:

* **self-contained**
  + This means it requires minimal support from the operating system or external library.
  + This makes SQLite usable in any environments especially in embedded devices like iPhones, Android phones, game consoles, handheld media players, etc.
  + SQLite is developed using ANSI-C.
* **server-less**
  + Usually RDBMS such as MySQL, PostgreSQL, etc., requires a separate server process to operate.
  + The applications that want to access the database server use TCP/IP protocol to send and receive requests.
  + This is called client/server architecture.

***(RDBMS client/server architecture)***

* + **SQLite does NOT require a server to run**.
  + SQLite database is integrated with the application that accesses the database.
  + The applications interact with the SQLite database read and write directly from the database files stored on disk.

***  
(SQLite server-less architecture)***

* **zero-configuration**
  + Because of the server-less architecture, there’s no need to “install” SQLite before using it.
  + There is no server process that needs to be configured, started, and stopped.
  + In addition, SQLite does not use any configuration files.
* **Transactional**
  + All transactions in SQLite are fully **ACID**-compliant.
    - means all queries and changes are **A**tomic, **C**onsistent, **I**solated, and **D**urable.
  + In other words, all changes within a transaction take place completely or not at all even when an unexpected situation like application crash, power failure, or operating system crash occurs.

### SQLite distinctive features

* SQLite uses dynamic types for tables.
  + It allows to store any value in any column, regardless of the data type.
* SQLite allows a single database connection to access multiple database files simultaneously.
  + This brings many nice features like joining tables in different databases or copying data between databases in a single command.
* SQLite is capable of creating in-memory databases which are very fast to work with.

## Android Database storage with SQLite

Android SDK comes with built in support for SQLite Database. Android SQLite Database is a light weight database mainly useful for embedded applications, it reads and writes directly to disk files.

* All the classes and interfaces that are required to work with Android SQLite Database are available in the package, “**android.database.sqlite**”.
* The “**SQLiteDatabase**” class represents a file on the Android device.
* Developer can control the name of the file which contains the DB and it will be a single file rather than multiple scattered files.

### Important classes used in Android SQLite

Some of the important classes which are used when working with an SQLite database listed below;

|  |  |
| --- | --- |
| Class | Description |
| SQLiteDatabase | represents the Android SQLite database |
| SQLiteOpenHelper | a helper class that manages the Android SQLite DB |
| SQLiteQuery | represents an Android SQLite DB query |
| SQLiteStatement | represents an Android SQLite statement |
| SQLiteCursor | Exposes the results from a query, use to iterate through the results from the query. |
| SQLiteQueryBuilder | a helper class to build and manage queries |
| SQLiteException | represents an Android SQLite Exceptions |

#### SQLiteOpenHelper Class

This class is designed to make it easier to create and update databases.   
Its methods include:

* **onCreate():** called when the database is created for the first time.
* **onUpgrade():** called in the event that the application code contains a more recent database version number reference.
* **onOpen():** called when the database is opened.
* **getWritableDatabase():** opens or creates a database for reading and writing.
* **getReadableDatabase**(): creates or opens a database for reading only.
* **close():** closes the database.

#### SQLiteDatabase Class

This class provides the primary interface between the application code and underlying SQLite database.  
Its methods include:

* **insert():** inserts a new row into a database table.
* **delete():** deletes rows from a database table
* **query():** performs a specified database query and returns matching results via a Cursor object.
* **execSQL():** executes a single SQL Statement that does not return result data.
* **rawQuery():** executes an SQL query statement and returns matching results in the form of a Cursor object.

#### Cursor class

This class provides access to the results of a database query.   
Its methods include:

* **close()**: release all resources used by cursor and close it.
* **getCount()**: returns the number of rows contained within the result set.
* **moveToFirst()**: moves to the first row in the result set.
* **moveToLast()**: moves to the last row in the result set.
* **moveToNext()**: moves to the next row in the result set.
* **move()**: moves by a specified offset from the current position in the result set.
* **get<type>()** (such as **getInt(),** **getDouble()**, so on): returns the value of the specified <type> contained at the specified column index of the row at the current cursor position.

#### ContentValues class

This class allows key/value pairs to be declared consisting of table column identifiers and the values to be stored in each column.   
Its methods include:

* **put()**: adds a value to the set.

## Saving messages sent in Database

With the theoretical knowledge above, lets save the messages we are sending through our application in a database, after the click on “SEND” button.

### The database table structure to save a message

First let’s sketch the design of the table in which a message is saved

Table Name:- “**message**”

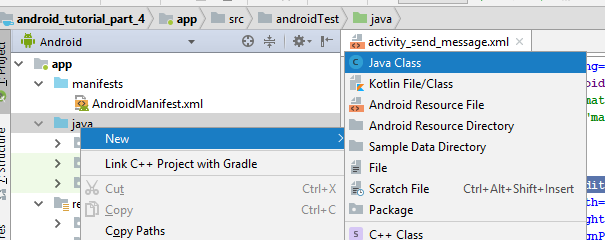
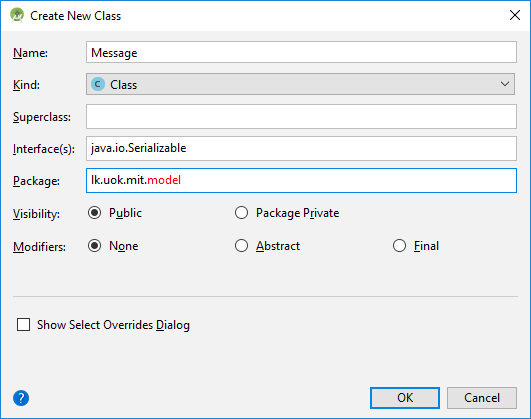
|  |  |  |  |
| --- | --- | --- | --- |
| Field Name | Field Type | Key | Comment |
| id | INTEGER | PRIMARY | An auto-incremented unique value for each record |
| contact\_number | TEXT |  | A filed to contain the contact number given as an input by user (the text from “**autoCompleteTextContactNumber**”) |
| message\_text | TEXT |  | A filed to contain the message given as an input by user (the text from “**editTextMessage**”) |
| sent\_time | TEXT |  | The current datetime value when user clicks “SEND” button |
| sent\_status | INTEGER |  | A status to record if the message could be sent or not |
| retry\_count | INTEGER |  | A value to keep count of how many attempts were made to send the message |

In addition to the contact number and message text input by the user, there are three additional fields in the above design.

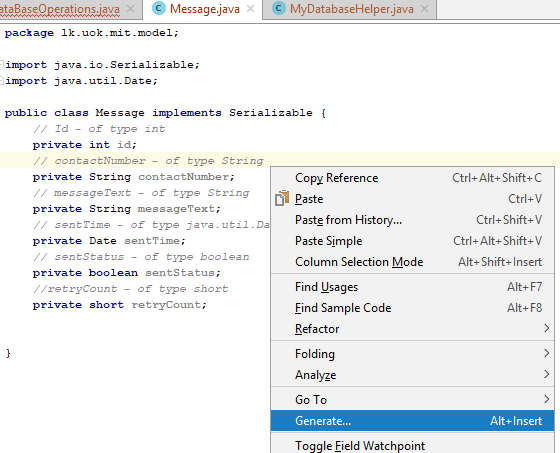
* **id**- auto - generated as an auto-increment at database level, can be used to uniquely identify a record
* **sent\_time** – the time at which user clicked “SEND” button, this is required when sorting from most recent message
* **sent\_status** – to indicate if the message could be sent or not
* **retry\_count –** this value is stored for a future use, in a next session, lets check how to try and resend the text messages which failed to be sent at first attempt.

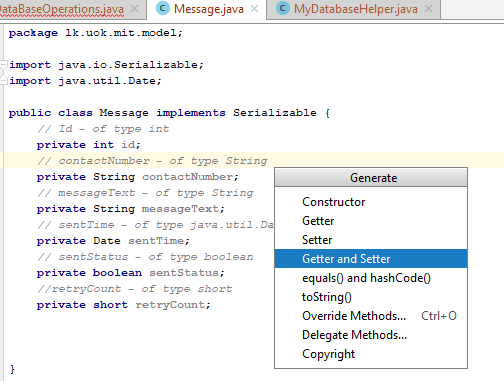
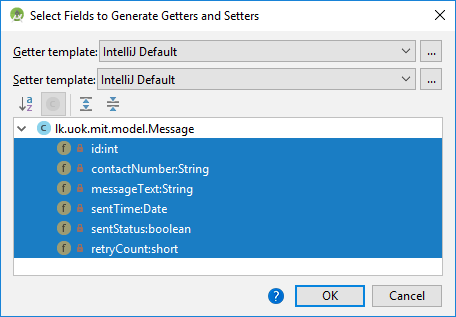
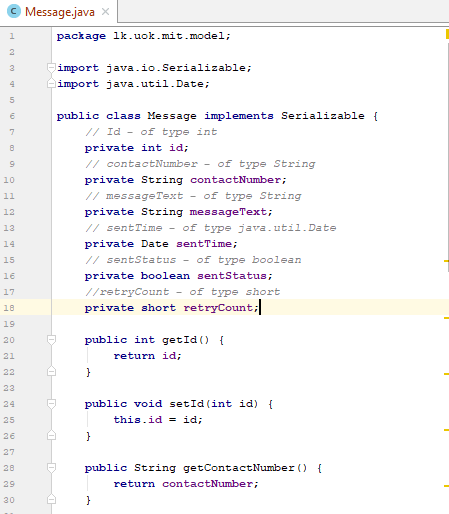
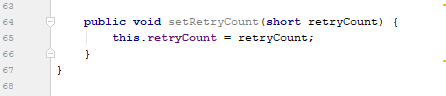
### Model Class for a Message

So far, we kept the contact number and message in two string variables when required for the app. But according to the table design above, you will notice there are four more fields to be stored. Now its time for us to create a model class to represent a message for easier coding.

1. Right click on “**java**” folder, and select “New”🡪”Java Class”  
   
2. Name the new class as “**Message**” inside a package called “**lk.uok.mit.model**” and implement “**Serializable**” interface  
   
3. Create below private member variables inside the class “**Message**” and create the **getter** and **setter** methods for them

* Id – of type int
  + **private int id**;
* contactNumber – of type String
  + **private** String **contactNumber**;
* messageText – of type String
  + **private** String **messageText**;
* sentTime – of type java.util.Date
  + **private** Date **sentTime**;
* sentStatus – of type boolean
  + **private boolean sentStatus**;
* retryCount – of type boolean
  + **private short retryCount**;
* After declaring the private member variables of the class, to generate the getter and setter methods,
  + Right click on the class (or press Alt+Insert) and select “**Generate**”



* + From the next drop down, select “getter and setter” option  
    
  + In the next screen, all the private variables of the “Message” class is listed, click on all of them while pressing “Ctrl” button on keyboard to select all fields, and then click “OK”  
    
* Now the complete “Message” class will look like below;  
    
    
  

### Mapping between relational DB table and java model class

Below is the object relational mapping of the relational table “message” and the domain java class “Message”

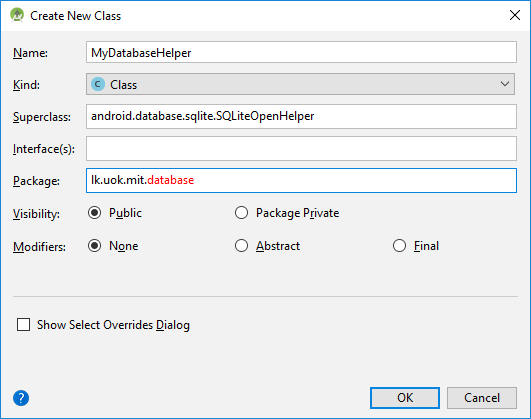
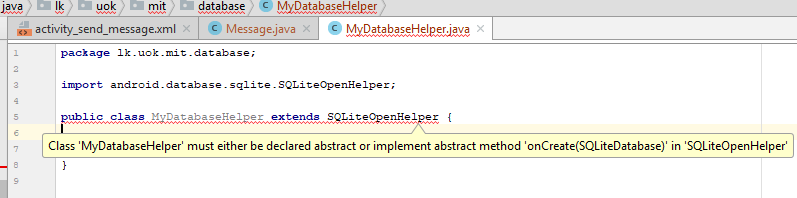
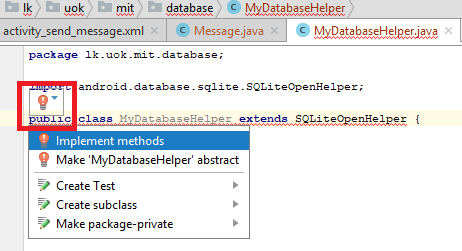
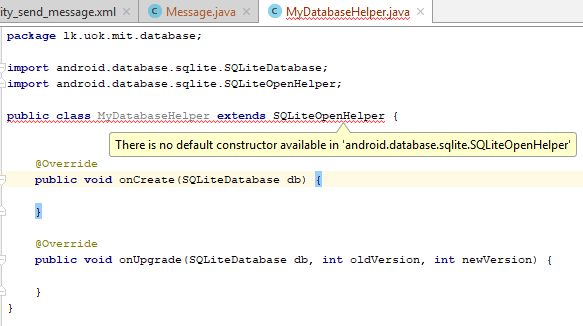
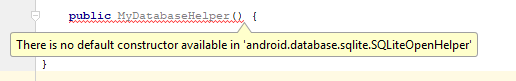
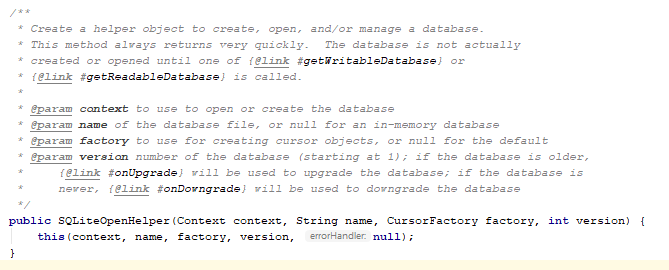
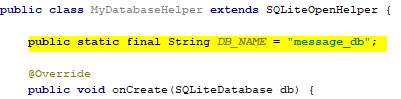
|  |  |  |  |
| --- | --- | --- | --- |
| Table “message” | | Java Model Class “Message” | |
| Field Name | Field Type | Field Name | Field type |
| id | INTEGER | id | int |
| contact\_number | TEXT | contactNumber | String |
| message\_text | TEXT | messageText | String |
| sent\_time | TEXT | sentTime | Java.util.Date |
| sent\_status | INTEGER | sentStatus | boolean |
| retry\_count | INTEGER | retryCount | short |

### SQLite Helper Class - MyDatabaseHelper

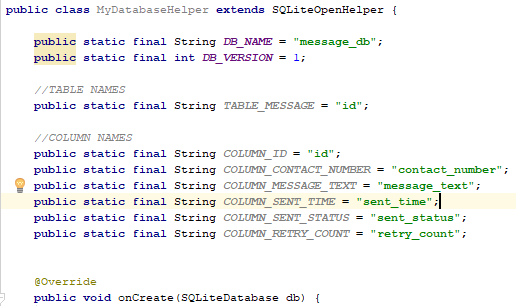
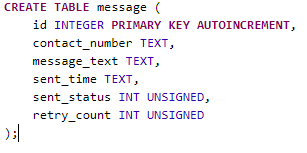
In order to work with SQLite database a utility helper class is required. For this purpose, let’s create a class names “**MyDatabaseHelper**”. This class should extends from “**SQLiteOpenHelper”** class.

There are two important methods that should be overridden as “**onCreate()**” and “**onUpgrade()**”.

* **onCreate()**
  + This method will be called **only once** when the **app is installed**
  + These is where to write **create table** statements.
  + This is called when database is created.
* **onUpgrade()**
  + This method will be called when an **update is released** to the app
  + in order to execute this method, modify the **DATABASE\_VERSION**
  + This method can be used for database upgraded like modifying the table structure, adding constraints to database, without losing existing data

1. Right click on “**java**” folder, and select “New”🡪”Java Class”
2. Name the new class as “**MyDatabaseHelper**” inside a package called “**lk.uok.mit.database**” and extend from “**android.database.sqlite.SQLiteOpenHelper**” class  
   
3. Initially there will be an en error in the new class as shown below, that’s because we have not overridden the “**onCreate**” or “**onUpgrade**” methods from the super class yet, or added a default constructor  
   
4. To fix this, click on the line with the error (underlines in red color) and hover it till a **red color bulb shaped tool tip** appears on top of that line, and click on the **arrow mark** on it as shown below, and click on “**implement methods**” in the listed menu  
   
   1. The above step will generate the method signatures for both “**onCreate**” and “**onUpgrade**” methods
5. But still there will be an error remaining as shown below, this is due to a missing constructor call to the super class, which means our super class “**android.database.sqlite.SQLiteOpenHelper**” does not have a default constructor, and therefore our sub class’s default constructor will not be enough.   
     
   1. To fix this lets add a constructor method to our class and call super class’s constructor method manually as shown below;  
      ***public MyDatabaseHelper() { }***  
      Now the error occurs at the constructor, to fix this we should call a constructor available in the super class, for that first let’s check the “**android.database.sqlite.SQLiteOpenHelper**” to know what parameters to pass in to its construtor
   2. If you check the super class “**android.database.sqlite.SQLiteOpenHelper**” (“ctrl + click” on class name) you will find different overloaded constructor methods in it, let’s use the following on in our case;  
        
      1. **context** - The app context which is to be used to open or create the database
         1. This parameter cannot be initialized from within our “**MyDatabaseHelper**” class since the app context is available in “Activity” classes for the app
         2. Therefore, this parameter should be passed in to the constructor of the “**MyDatabaseHelper**” from the caller
         3. So far the method signature looks like below;  
            **public** MyDatabaseHelper(Context context) {  
            }
      2. **name** - The name of the database file, or null for an in-memory database
         1. Since the database creation and upgrade is done through this helper class, this parameter should be initialized inside the “**MyDatabaseHelper**” as a **public static final** class level constant of type String with name as “**DB\_NAME**” and value as “**message\_db**”
            1. if any other class required to use it, it should be visible – hence “**public**”
            2. since it’s a constant, no need to make it an instance variable – hence “**static**”
            3. The name of the database will not be changed dynamically – hence “**final**”  
               **public static final** String ***DB\_NAME*** = **"message\_db"**;
      3. **factory** - A cursor factory to use for creating cursor objects, or null for the default
         1. We don’t need a custom cursor factory, the default one is adequate and therefore we can pass “**null**” for this
      4. **version** – The version number of the database (starting at 1)
         1. The version number is important when publishing updates to the same application and updates contains DB changes.
         2. As same as the database name, a constant named “**DB\_VERSION**” should be declared at class level of type int marked as a constant by using **public static final** modifiers  
            **public static final int *DB\_VERSION*** = 1;
      5. After completing above steps, now we can simply call the super class’s constructer by passing parameters as mentioned above  
         **public** MyDatabaseHelper(Context context) {  
          **super**(context, ***DB\_NAME***, **null**, ***DB\_VERSION***);  
         }
      6. Now you will notice all the compile time errors on the class has gone at this point
   3. Now it’s time to create the database; recall from above, to database creation, the overridden method “onCreate” is be used. Currently the method looks like below;  
      @Override  
      **public void** onCreate(SQLiteDatabase db) {  
       *//execute the CREATE TABLE STATEMENT*}
   4. A parameter is passed in to this method of type “**android.database.sqlite.SQLiteDatabase**”
   5. This database is created and available in this method, based on the parameters we passed in to the super class’s constructor as demonstrated above
   6. Inside this “**onCreate()**” method we are permitted to execute SQL statements using the “**execSQL()”** method of the “**db**” parameter.
      1. “**execSQL()”** - Execute a single SQL statement that is NOT a SELECT or any other SQL statement that returns data.
      2. Accepts a parameter of type String – should be the SQL statement to be executed
   7. In order to pass the SQL statement to create the table “**message**”, it should be constructed based on the design we did earlier
   8. Construct the String of “CREATE table” statement for “**message**” based on the given design and when constructing the string;  
        
        
        
      1. To recall, below is the table to be created

|  |  |  |
| --- | --- | --- |
| Field Name | Field Type | Key |
| id | INTEGER | PRIMARY |
| contact\_number | TEXT |  |
| message\_text | TEXT |  |
| sent\_time | TEXT |  |
| sent\_status | INTEGER |  |
| retry\_count | INTEGER |  |

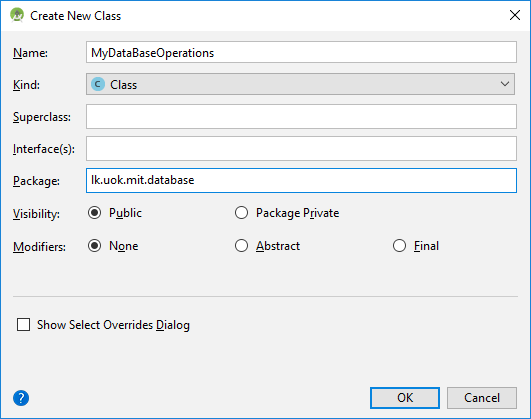
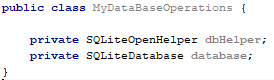
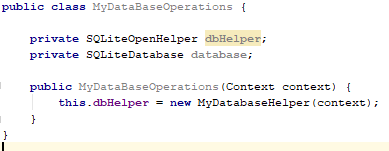
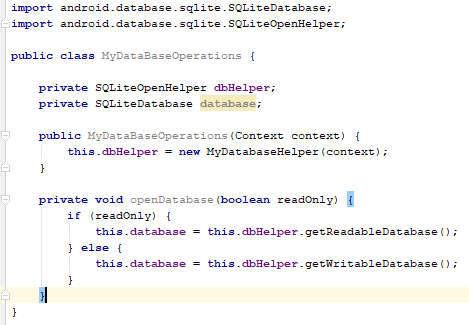
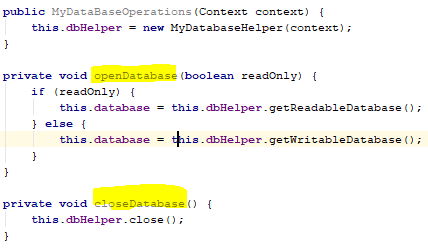
* + 1. Define column names as class level public String constants (these are used for select statements as well
       1. TABLE NAMES  
          **public static final** String ***TABLE\_MESSAGE*** = **"message"**;
       2. COLUMN NAMES  
          **public static final** String ***COLUMN\_ID*** = **"id"**;  
          **public static final** String ***COLUMN\_CONTACT\_NUMBER*** = **"contact\_number"**;  
          **public static final** String ***COLUMN\_MESSAGE\_TEXT*** = **"message\_text"**;  
          **public static final** String ***COLUMN\_SENT\_TIME*** = **"sent\_time"**;  
          **public static final** String ***COLUMN\_SENT\_STATUS*** = **"sent\_status"**;  
          **public static final** String ***COLUMN\_RETRY\_COUNT*** = **"retry\_count"**;
       3. The code will look like below at this point, note the “TABLE” prefix for table names and “COLUMN” prefix for columns
    2. Construct the “CREATE\_TABLE” SQL String using “**StringBuilder**” class in Java,
       1. in plain SQL the statement should look like below  
          
       2. When the above statement is Constructed as a string in Java using “**StringBuilder**” class, it looks like below, (here we are using the column names defined as constants in earlier step)  
          **public static final** String ***SQL\_CREATE\_TABLE\_MESSAGE*** =  
           **new** StringBuilder(**"CREATE TABLE "**).append(***TABLE\_MESSAGE***).append(**" ( "**)  
           .append(***COLUMN\_ID***).append(**" INTEGER PRIMARY KEY AUTOINCREMENT "**).append(**","**)  
           .append(***COLUMN\_CONTACT\_NUMBER***).append(**" TEXT "**).append(**","**)  
           .append(***COLUMN\_MESSAGE\_TEXT***).append(**" TEXT "**).append(**","**)  
           .append(***COLUMN\_SENT\_TIME***).append(**" TEXT "**).append(**","**)  
           .append(***COLUMN\_SENT\_STATUS***).append(**" INT UNSIGNED "**).append(**","**)  
           .append(***COLUMN\_RETRY\_COUNT***).append(**" INT UNSIGNED "**)  
           .append(**" ) "**).toString();  
          **Note the spaces before and after every string value appended**
    3. After constructing the “CREATE TABLE” statement in java as a string and assigning it to a constant named “**SQL\_CREATE\_TABLE\_MESSAGE**”, now its time to call “**execSQL()**” method from inside “onCreate()” method of the “**MyDatabaseHelper**” class as shown below;  
       @Override  
       **public void** onCreate(SQLiteDatabase db) {  
        *//execute the CREATE TABLE STATEMENT* db.execSQL(***SQL\_CREATE\_TABLE\_MESSAGE***);  
       }
    4. This concludes database creation part
  1. Now only “**onUpgrade**” method is left unimplemented in the “**MyDatabaseHelper**” class.
     1. Ideally, this method should be used for database changes upon app version updates
     2. For the sake of completion, in our example, lets “drop” the “message” table and then “create” it for now,
     3. Construct the SQL statement to DROP TABLE as a String using “**StringBuilder**” class in Java
        1. in plain SQL the statement should look like below  
           
        2. After constructing it as a String and assigning to a constant named “SQL\_DROP\_TABLE\_MESSAGE” using “**StringBuilder**” class in Java it look s like below;  
           **public static final** String ***SQL\_DROP\_TABLE\_MESSAGE*** =  
            **new** StringBuilder(**"DROP TABLE IF EXISTS "**).append(***TABLE\_MESSAGE***).toString();
        3. Now lets call “**execSQL()“** from inside “**onUpgrade()**” method for both drop and create SQL statements like below;  
           @Override  
           **public void** onUpgrade(SQLiteDatabase db, **int** oldVersion, **int** newVersion) {  
            db.execSQL(***SQL\_DROP\_TABLE\_MESSAGE***);  
            db.execSQL(***SQL\_CREATE\_TABLE\_MESSAGE***);  
           }
     4. This means, if the current value of **DB\_VERSION -> 1** changes for some reason, the “message” table will be dropped with any data it had, and will be created again.

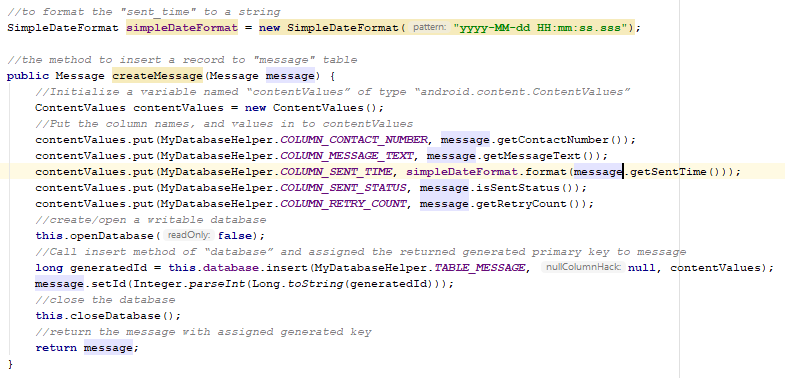
1. With that, the implementetaion of our Database Helper class named “**MyDatabaseHelper**” is completed

### DB operation helper class – MyDataBaseOperations

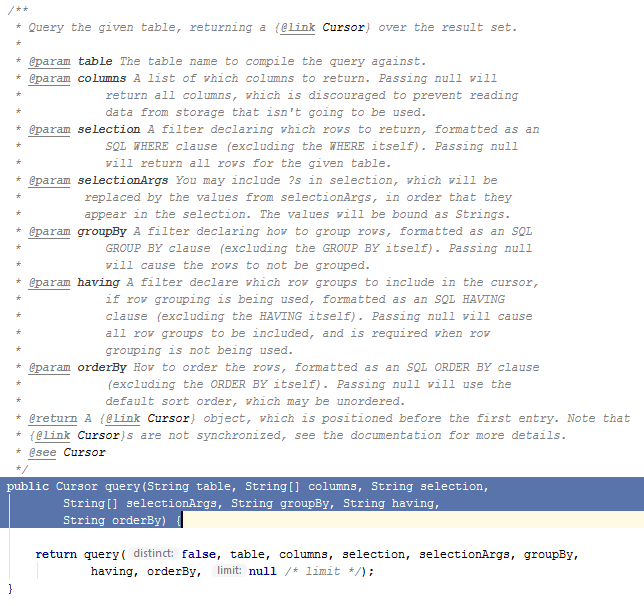
This class holds the database related methods to perform the CRUD operations - **C**reate, **R**ead, **U**pdate and **D**elete operation for messages

Unlike the “**MyDatabaseHelper**” class we created above, which extended from “**android.database.sqlite.SQLiteOpenHelper**”, this “**MyDataBaseOperations**” is a custom class we are going to write by ourselves.

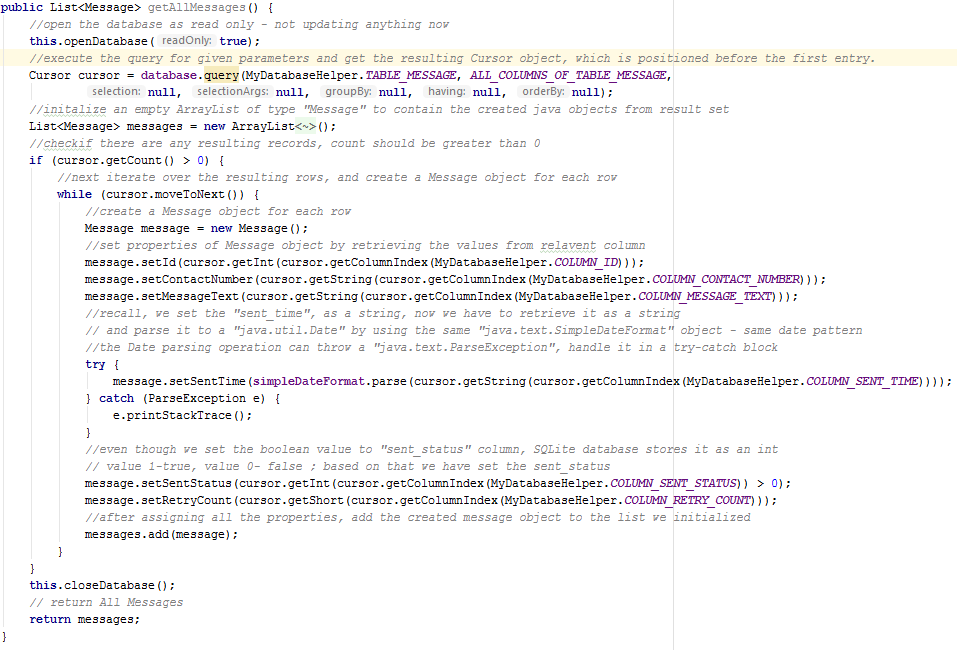
1. Right click on “**java**” folder, and select “New”🡪”Java Class”
2. Name the new class as “**MyDataBaseOperations**” inside the package “**lk.uok.mit.database**”   
   
3. In order to read/write from “message” table, we need a reference to the “**SQLiteDatabase**” class, which provides the SQL statement execution facilities,   
   At the same time, to get a “**SQLiteDatabase**” instance, we need a reference to “**SQLiteOpenHelper**” Class, which has “**getWritableDatabase()**” and “**getReadableDatabase()**” methods
   1. First add two class level properties to hold these two variables  
      **private** SQLiteOpenHelper **dbHelper**;   
      **private** SQLiteDatabase **database**;  
      
   2. We have our own “**SQLiteOpenHelper**” implementation named “**MyDatabaseHelper**”, let’s initialize the “**dbHelper”** variable inside the **constructor** of this class  
      **public** MyDataBaseOperations(Context context) {  
       **this**.**dbHelper** = **new** MyDatabaseHelper(context);  
      }  
      
      1. Idea is, upon the initialization of “**MyDataBaseOperations**”, we need an instance of “**MyDatabaseHelper**”.
      2. After that during method executions, we can get database through our “**dbHelper**”
      3. We need to accept a “Context” variable through our constructer, since the “**MyDatabaseHelper**” class’s constructor requires it.
4. Next, lets write a method to get a readable or writable database assigned to our “**database**” variable based on the boolean value we pass
   1. Method Name :- openDatabase ()
   2. Method Return Type:- void
   3. Method Parameters:- boolean readOnly --> (if a read-only db or not)
   4. Method Access Modifier:- private
   5. Method Logic: -   
      Check If its requires a read-only database or not, and   
      if read-only🡪call getReadableDatabase() of “**dbHelper**” to create or open a database for reading only.  
      else 🡪 call getWritableDatabase() of “**dbHelper**” to open or create a database for reading and writing.  
        
      **private void** openDatabase(**boolean** readOnly) {  
       **if** (readOnly) {  
       **this**.**database** = **this**.**dbHelper**.getReadableDatabase();  
       } **else** {  
       **this**.**database** = **this**.**dbHelper**.getWritableDatabase();  
       }  
      }
   6. The “**MyDataBaseOperations**” class looks like below now,  
      
5. Next a method should be written to close the database
   1. Method Name :- closeDatabase ()
   2. Method Return Type:- void
   3. Method Parameters:- none
   4. Method Access Modifier:- private  
      Method Logic: - Call the “**close()**” method of “**SQLiteOpenHelper**” class available to us through “**dbHelper**” (instance of “**MyDatabaseHelper**”)  
        
      **private void** closeDatabase(){  
       **this**.**dbHelper**.close();  
      }
   5. Up to now below constructer and highlighted methods are available in our class  
      
6. Next, let’s write a method to save a message to our database.
   1. Method Name :- createMessage ()
   2. Method Return Type:- **lk.uok.mit.model.Message**
   3. Method Parameters:- message 🡪 of type **lk.uok.mit.model.Message**
   4. Method Access Modifier:- public  
      Method Logic: -
      1. Initialize a variable named “**contentValues**” of type “**android.content.ContentValues**”  
         ***ContentValues contentValues = new ContentValues()***;
      2. Put the column names, and values in to **contentValues  
         *contentValues.put(MyDatabaseHelper.COLUMN\_CONTACT\_NUMBER, message.getContactNumber());  
         contentValues.put(MyDatabaseHelper.COLUMN\_MESSAGE\_TEXT, message.getMessageText());  
         contentValues.put(MyDatabaseHelper.COLUMN\_SENT\_TIME, simpleDateFormat.format(message.getSentTime()));  
         contentValues.put(MyDatabaseHelper.COLUMN\_SENT\_STATUS, message.isSentStatus());  
         contentValues.put(MyDatabaseHelper.COLUMN\_RETRY\_COUNT, message.getRetryCount());***
         1. For column names, use the defined constants in“**MyDatabaseHelper**” class
         2. For the values, use getter methods of he passed “message” object
         3. When using the getter of “**SentTime**” filed, use a “**java.text.SimpleDateFormat**” to format the “**java.util.Date**” object to an **ISO8601 string format** of the date time – this is because SQLite does not support date time data types  
            ***SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.sss");***
      3. Open the database in writable mode  
         ***this.openDatabase(false);***
      4. Call insert method of **“database”** and assigned thereturned **generated primary key** to **message  
         *long insertid = this.database.insert(MyDatabaseHelper.TABLE\_MESSAGE, null, contentValues);  
         message.setId(Integer.parseInt(Long.toString(insertid)));***
      5. Close the database  
         ***this.closeDatabase();***
      6. Return message with generated primary key  
         ***return message;***
   5. The whole method will look like below;



1. Write a method to retrieve all the messages from table “message”
   1. Method Name :- getAllMessages ()
   2. Method Return Type:- a “**java.util.List**”of type **lk.uok.mit.model.Message**
   3. Method Parameters:- none
   4. Method Access Modifier:- public  
      Method Logic: -
      1. There are multiple ways to write a select query using the “**database**” (instance of “**SQLiteDatabase**”)
         1. Using “**rawQuery()**” method by passing a native SELECT query
         2. Using “**query()**” method by passing table name, column names to be selected and other arguments for WHERE clause, HAVING clause and GROUP BY clause
      2. We will use the second approach here, that is to use the “query()” method of the “**SQLiteDatabase**”) class
      3. The method signature of the “**query()**” method in “**android.database.sqlite.SQLiteDatabase**” class is shown below;



* + 1. Since we are **retrieving all records** form the “message” tables, we only have to pass parameters of “table name” and “columns”.   
       Other parameters for **WHERE** clause, **GROUP BY** clause, **HAVING** clause and **ORDER BY** clause can be skipped – **pass null** for them
    2. As shown above, We have to pass an array of column names we need to retrieve, lets declare it as a **class level constant** first;  
       ***private static final String[] ALL\_COLUMNS\_OF\_TABLE\_MESSAGE = {  
        MyDatabaseHelper.COLUMN\_ID,  
        MyDatabaseHelper.COLUMN\_CONTACT\_NUMBER,  
        MyDatabaseHelper.COLUMN\_MESSAGE\_TEXT,  
        MyDatabaseHelper.COLUMN\_SENT\_TIME,  
        MyDatabaseHelper.COLUMN\_SENT\_STATUS,  
        MyDatabaseHelper.COLUMN\_RETRY\_COUNT  
       };***
    3. Now, as the first line of our method, open the database in read only mode, since we do not do any updates to the database in this method  
       ***this.openDatabase(true);***
    4. Next query the database by passing **database name** and **column array** and get the resulting “**android.database.Cursor**” to a variable  
       ***Cursor cursor = database.query(MyDatabaseHelper.TABLE\_MESSAGE, ALL\_COLUMNS\_OF\_TABLE\_MESSAGE,  
        null, null, null, null, null);***
    5. Next, initialize an empty ArrayList of type "Message" to contain the created java objects from result set  
       ***List<Message> messages = new ArrayList<Message>();***
    6. Before doing any operation on resulted “**cursor**”, check if there are any resulting records, count should be greater than 0  
       ***if (cursor.getCount() > 0) {***
    7. If results are available, iterate over the resulting rows, and create a Message object for each row  
       ***while (cursor.moveToNext()) {***
    8. Create a Message object for each row  
       ***Message message = new Message();***
    9. Next for all the columns; set properties of Message object by retrieving the values from relevant column  
       ***message.setId(cursor.getInt(cursor.getColumnIndex(***MyDatabaseHelper.***COLUMN\_ID)));***
       1. As it was mentioned in above sections as well, there’s a method in Cursor class named **get<type>()**   
          e.g:- such as **getInt(),** **getDouble(),** **getString(), etc.**  
          It returns the value of the specified <type> contained at the specified **column index** of the row at the current cursor position.
       2. To call those getInt()/getString() like methods, we do not have the column indexes of the result set with us,   
          as a solution, there is another method in cursor class as “**getColumnIndex()**”, which accepts the column’s name and returns its index
       3. Therefore,   
          ***cursor.getColumnIndex(MyDatabaseHelper.COLUMN\_ID)*** -> returns column’s index  
          ***cursor.getInt(COLUMN\_INDEX)***-> returns the column’s value at the current row as an Integer  
          ***message.setId(ID) ->*** sets the passed values as id of the message
       4. Same as above, rest of the columns values of the current row of cursor can also be retrieved and set to the message;
       5. There is some more work to be done when setting the “SentTime” which is of type “**java.util.Date**”  
          **try** { message.setSentTime(**simpleDateFormat.parse(***cursor.getString*(cursor.getColumnIndex(MyDatabaseHelper.***COLUMN\_SENT\_TIME***)*)***)**);  
           } **catch** (ParseException e) {  
           e.printStackTrace();  
           }  
            
          Recall, we set the "**sent\_time**", as a string, now we have to retrieve it as a **String**, and parse it to a "**java.util.Date**" by using the same "**java.text.SimpleDateFormat**" object (same date pattern)  
          The Date parsing operation can throw a "java.text.ParseException", handle it in a try-catch block
       6. Setting the “SentStatus” of type Boolean is also requires some additional work;  
          even though we set the **boolean** value to "**sent\_status**" column, SQLite database stores it as an **int** value.  
          **value 1-true,** **value 0- false** ; based on that we have set the sent\_status as shown below;  
          message.setSentStatus(***cursor.getInt(cursor.getColumnIndex(MyDatabaseHelper.COLUMN\_SENT\_STATUS))*** > 0);
    10. The complete method will look like below;



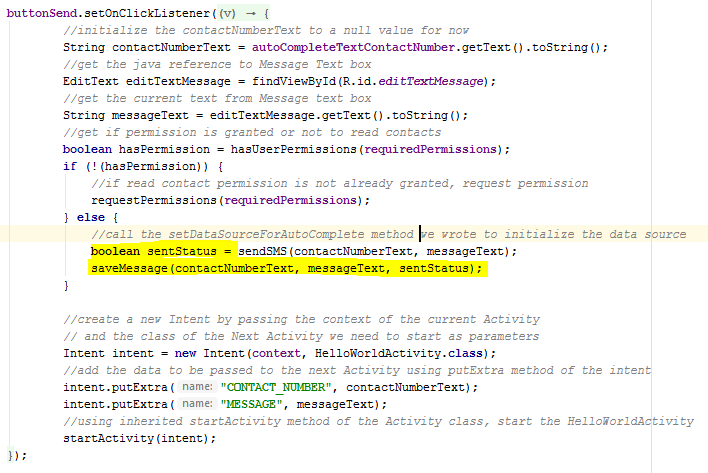
1. With the two methods mentioned below, the DB operations helper class (“**MyDataBaseOperations**”) is completed
   1. createMessage() – to insert a message to database
   2. getAllMessages() – to retrieve all the messages from database

### Call createMessage() method from “SendMessageActivity” class

To call methods of “**MyDataBaseOperations**” class from inside “**SendMessageActivity**” we will need a reference to an instance of that type. Therefore lets create an instance of db operations helper class and call its **createMesssage()** method to save the sent message on “**SEND**” button’s click method, as shown below;

1. Open “**lk.uok.mit.activity**.**SendMessageActivity**” inside “**java**” folder
2. Create a class level variable with name “**dataBaseOperations**” of type “**lk.uok.mit.database.MyDataBaseOperations**”  
   ***private MyDataBaseOperations dataBaseOperations;***
3. Inside “onCreate” method, initiate the instance by passing context to the constructor  
   ***this.dataBaseOperations = new MyDataBaseOperations(context);***
4. Create a method to save the message, inside this we will call the “***MyDataBaseOperations***” class’s “**createMessage()**” method
   1. Method Name :- saveMessage ()
   2. Method Return Type:- void
   3. Method Parameters:-
      1. String contactNumber
      2. String messageText
      3. boolean sentStatus
   4. Method Access Modifier:- private  
      Method Logic: -
      1. Create a new “**lk.uok.mit.model.Message**” object  
         ***Message message = new Message();***
      2. Set he passes parameters to the new object as shown below;  
         ***message.setContactNumber(contactNumber);  
         message.setMessageText(messageText);  
         message.setSentStatus(sentStatus);***
      3. Set the current time as sentTime  
         ***message.setSentTime(Calendar.getInstance().getTime());***
      4. Set 1 as the retry count  
         ***message.setRetryCount((short) 1);***
      5. Call the “**createMessage()**” method of ***MyDataBaseOperations*** *class*

***this.dataBaseOperations.createMessage(message);***

1. Now we need to call the “**saveMessage()”** method from within onclick method of “**SEND**” button, after calling “**sendSMS**” method
2. Before that, we need to change the “**sendSMS**” method to return a **boolean** value like shown below;  
   ***private boolean sendSMS(String contactNumber, String message) {  
    boolean sentStatus = true;  
    //include the code snippet in a try-catch block as this operation is likely to throw errors  
    try {  
    //i. Get a reference to “android.telephony.SmsManager” class  
    SmsManager smsManager = SmsManager.getDefault();  
    //use the sendTextMessage method to send the SMS and pass the contact number and message as parameters  
    smsManager.sendTextMessage(contactNumber, null, message, null, null);  
    //if code reached here, it means no exceptions occurred, notify the user of success  
    Toast.makeText(this.context, "SMS Sent!", Toast.LENGTH\_LONG).show();  
    } catch (Exception e) {  
    //if an exception occurs, set the sent status as false  
    sentStatus = false;  
    //if code reached here, it means an exception has occurred, notify the user of failure  
    Toast.makeText(this.context, "SMS faild, please try again later!", Toast.LENGTH\_LONG).show();  
    //print the error trace to fins the cause  
    e.printStackTrace();  
    }  
    return sentStatus;  
   }***
3. call the “**saveMessage()”** method from within onclick method of “**SEND**” button, after calling “**sendSMS()**”  
   ***boolean sentStatus = sendSMS(contactNumberText, messageText);  
   saveMessage(contactNumberText, messageText, sentStatus);***
4. The whole method will look like below; with this, saving part is completed!  
   

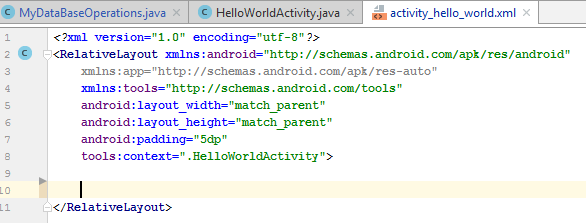
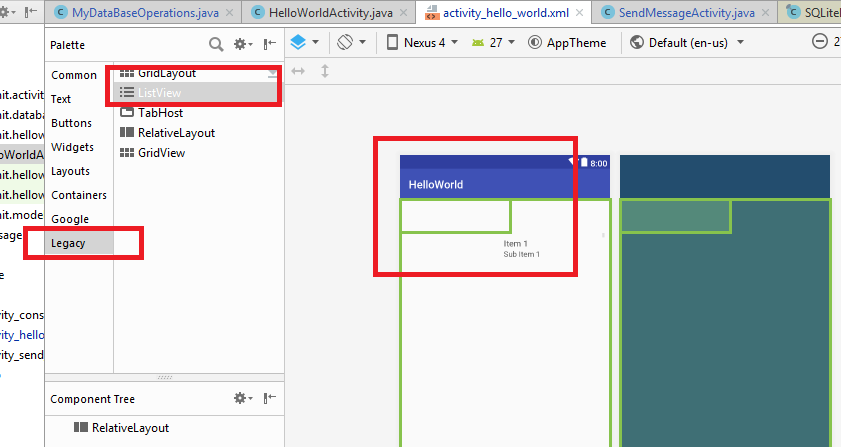
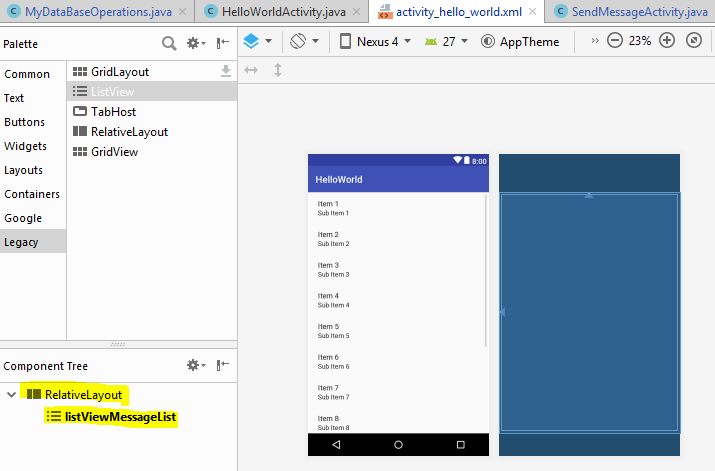
## Retrieving saved messages from Database

We already write a method inside the “**MyDataBaseOperations**” class with name “**getAllMessages**”, to retrieve all the messages from the database. Now lets see how to call that method, get the list and display in a list view.

We were using the “**HelloWorldActivity**” class so far to display the sent message. Lets modify the same activity class, and its layout (**activity\_hello\_world.xml**) to fetch the whole list from database and display.

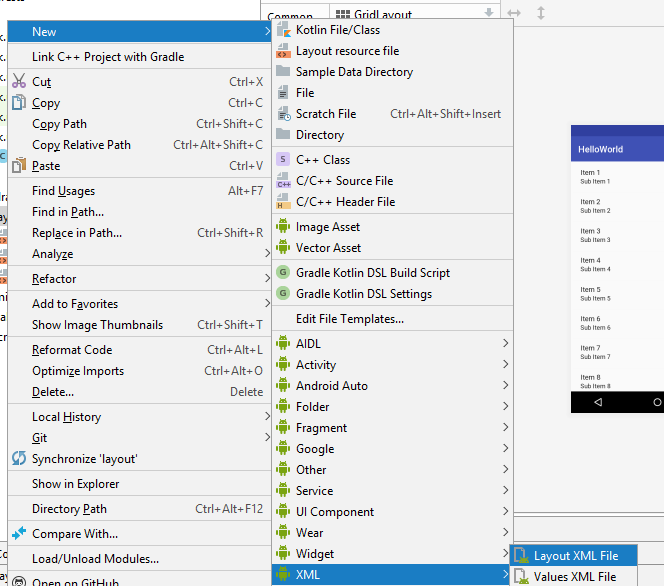
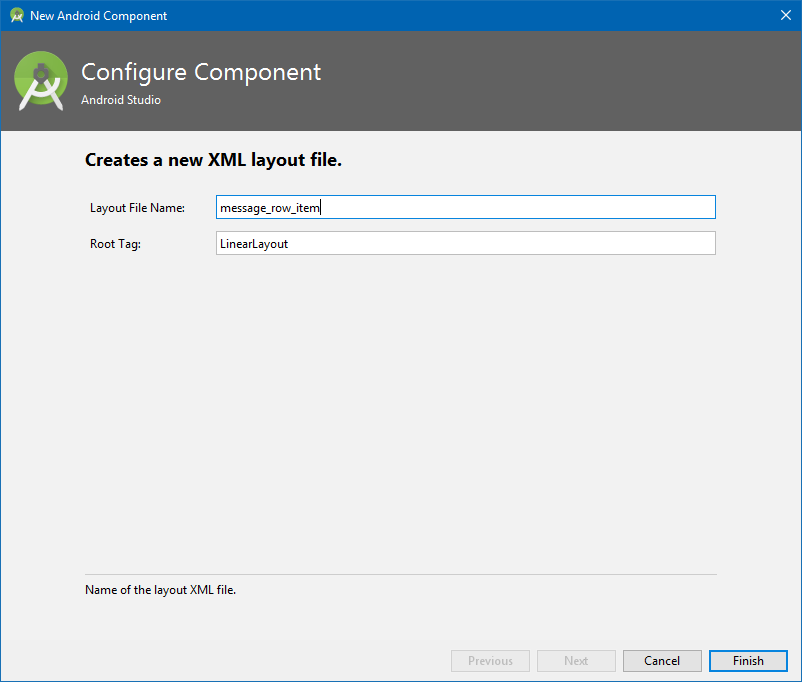
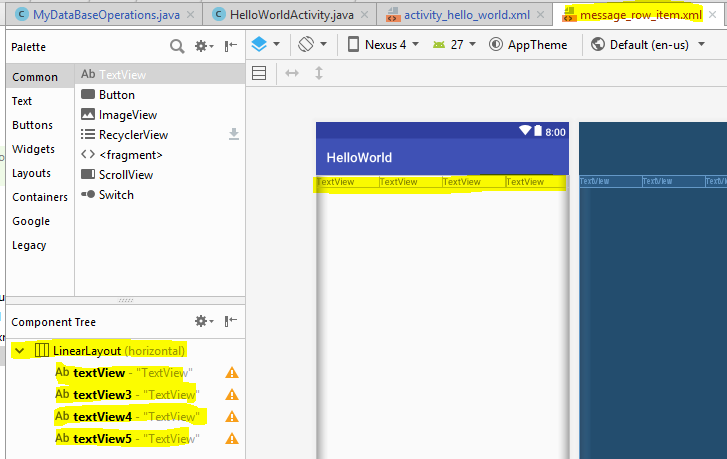
### Design parent ListView

To display the Message list, lets convert the “**activity\_hello\_world.xml**” in to a list view

1. Open “**activity\_hello\_world.xml**” file inside “**res/layout**”
2. Go to the “Text” view and delete both “**TextView**” s from the code
3. Only the <**RelativeLayout**> tags should be available like shown below;  
   
4. Now go to the “Legacy” view and drag and drop a “**ListView**” from the “**Palette**”  
   
5. Now go again in to the “Text” view and modify the “**ListView**” as shown below;  
   <**ListView  
    android:id="@+id/listViewMessageList"  
    android:layout\_width="match\_parent"  
    android:layout\_height="wrap\_content"  
    android:layout\_alignParentStart="true"  
    android:layout\_alignParentLeft="true"  
    android:layout\_alignParentTop="true"** />
6. The design view looks like below now;  
   

### Design Single Row Item

As the next step, lets design the view of a single item of a row, that is which attributes to be displayed inside a single row

1. Go to the “layout” folder, right click and go to New-->XML--> Layout XML File  
   
2. Fill the fields like shown below; and click   
   
   1. Layout File Name is:- ***message\_row\_item***
   2. Root Tag:- ***LinearLayout***
3. Drag and drop four “**Textview**” views in to the “**Design**” view of the “***message\_row\_item.xml***”  
   
4. Now go to the “Text” view and change the properties of each tag as shown below;
   1. <**LinearLayout :-**
      1. Change android:layout\_height to :- **android:layout\_height="wrap\_content"**
      2. Add padding top and bottom as 5dp (will leave enough space between two rows of message list)
   2. Change the <TextViews and add **android:id,** **android:text,** and **android:layout\_weight** attributes as below

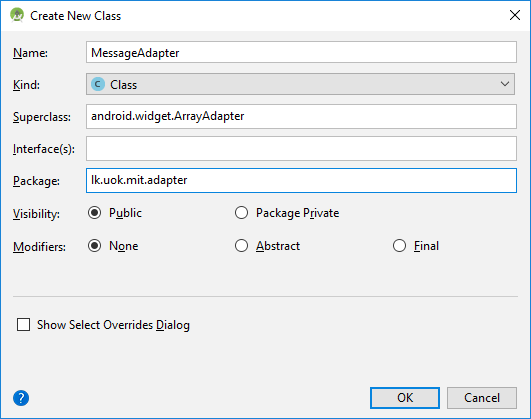
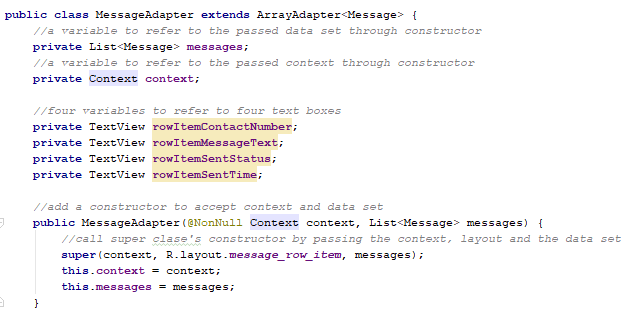
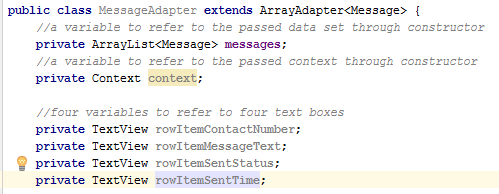
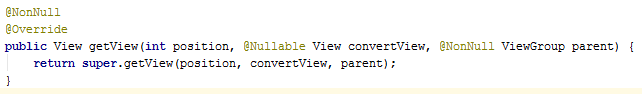
|  |  |  |
| --- | --- | --- |
| **android:id** | **android:text** | **android:layout\_weight** |
| textViewMessageRowItemContactNumber | Contact Number | 2 |
| textViewMessageRowItemMessageText | Message Text | 3 |
| textViewMessageRowItemSentStatus | Sent Status | 1 |
| textViewMessageRowItemSentTime | Sent Time | 2 |

* 1. Make text color black for all text views - **#00000**
  2. Make the width of all text views 0px – with a weight values specified, it means they will share width based on the weightage provided.
  3. The complete layout will look like below in text view  
     *<?***xml version="1.0" encoding="utf-8"***?>*<**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
      android:paddingTop="5dp"  
      android:paddingBottom="5dp"  
      android:layout\_width="match\_parent"  
      android:layout\_height="wrap\_content"**>  
       
      <**TextView  
      android:id="@+id/textViewMessageRowItemContactNumber"  
      android:layout\_width="0px"  
      android:layout\_height="wrap\_content"  
      android:layout\_weight="2"  
      android:textColor="#000000"  
      android:text="Contact Number"** />  
       
      <**TextView  
      android:id="@+id/textViewMessageRowItemMessageText"  
      android:layout\_width="0px"  
      android:layout\_height="wrap\_content"  
      android:layout\_weight="3"  
      android:textColor="#000000"  
      android:text="Message Text"** />  
       
      <**TextView  
      android:id="@+id/textViewMessageRowItemSentStatus"  
      android:layout\_width="0px"  
      android:layout\_height="wrap\_content"  
      android:layout\_weight="1"  
      android:textColor="#000000"  
      android:text="Sent Status"** />  
       
      <**TextView  
      android:id="@+id/textViewMessageRowItemSentTime"  
      android:layout\_width="0px"  
      android:layout\_height="wrap\_content"  
      android:layout\_weight="2"  
      android:textColor="#000000"  
      android:text="Sent Time"** />  
       
     </**LinearLayout**>

1. Now the UI design part is completed, next lets see the related java code

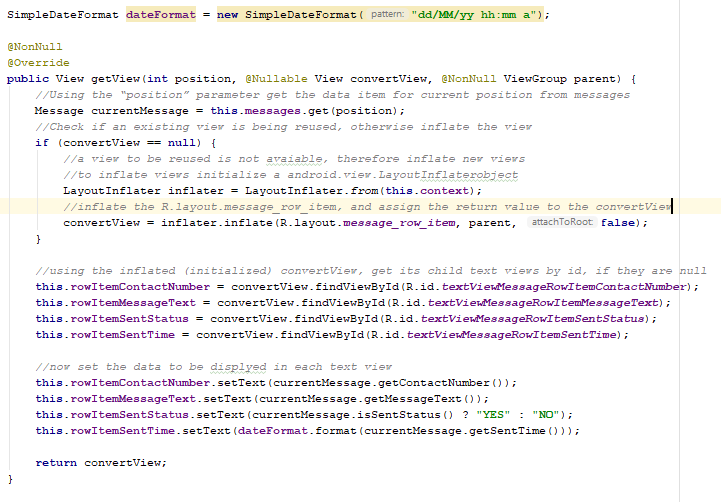
### Write a Custom Array Adapter

Let’s create a class named “**MessageAdapter**.java”. This adapter class is to render the **ListView** with defined layout (***message\_row\_item.xml***) and data set (**List<Message>**). The custom adapter class should extend from “**ArrayAdapter**” class.

1. Right click on “**java**” and got to “New”-->”Java Class”
2. Fill he details as shown below and create the custom array adapter named “**MessageAdapter**”  
   
   1. Class Name:- **MessageAdapter**
   2. Super class:- **android.widget.ArrayAdapter**
   3. Package:- **lk.uok.mit.adapter**
3. Go to the “**lk.uok.mit.adapter.MessageAdapter**” class in “java” folder and fix the compile error by adding a constructor as shown below in yellow color section;  
     
   ***public class MessageAdapter extends ArrayAdapter<Message> {  
    //a variable to refer to the passed data set through constructor  
    private ArrayList<Message> messages;  
    //a variable to refer to the passed context through constructor  
    private Context context;  
    //add a constructor to accept context and data set  
    public MessageAdapter(@NonNull Context context, List<Message> messages) {  
    //call super clase's constructor by passing the context, layout and the data set  
    super(context, R.layout.message\_row\_item, messages);  
    this.context = context;  
    this.messages = messages;  
    }  
   }***
   1. As shown in green color highlighted section, add the “**lk.uok.mit.model.Message**” as a type parameter to the “**ArrayAdapter**” class
   2. As shown in blue color section, declare two class variables to refer to the context and dataset passed in to the constructor as parameters
   3. The super class’s (**ArrayAdapter**) constructer expects the view’s id and the context
      1. Pass the context which is passed to the constructor of the “**MessageAdapter**” class
      2. Pass the resource id of the newly created layout to display a single message item :- “**R.layout.message\_row\_item**”
      3. In addition to the context parameter, from the constructor, get the dataset in to the adapter
      4. Assign the context and the dataset in to the class level variables we declared
      5. So far the code looks like below;  
         
4. Declare four class level variables of type “TextView” to refer to the four text views we added to the “**message\_row\_item**” layout  
   *//four variables to refer to four text boxes***private** TextView **rowItemContactNumber**;  
   **private** TextView **rowItemMessageText**;  
   **private** TextView **rowItemSentStatus**;  
   **private** TextView **rowItemSentTimel**;  
     
   
5. Next, override the “**getView()”** method of the superclass, here we can set each of the above text views value, **getView()** method isgoing to be called for each position every time it is displayed  
   
   1. For the overridden method, three parameters are passed as shown above
      1. **position**: - position of the item (index)
      2. **convertView**:- As **getView()** is call many times, inflating a new view every time is expensive so list view provides one of the previously created view to re-use  
         here our convert view will be the **message\_row\_item**
      3. **parent:-** A reference to the parent view that this view will be a child of.
   2. Let’s change the overridden method as indicated below to display the fetched values from db;
      1. Using the “position” parameter get the data item for current position  
         ***Message currentMessage = this.messages.get(position);***
      2. Check if an existing view is exist to be reused, otherwise inflate the view  
         ***if (convertView == null) {  
          //a view to be reused is not available, therefore inflate new views  
          //to inflate views initialize a android.view.LayoutInflaterobject  
          LayoutInflater inflater = LayoutInflater.from(this.context);  
          //inflate the R.layout.message\_row\_item, and assign the return value to the convertView  
          convertView = inflater.inflate(R.layout.message\_row\_item, parent, false);  
         }***
      3. using the inflated (initialized) convertView, get its child text views by id, and assign to the class variables declared above  
         ***this.rowItemContactNumber = convertView.findViewById(R.id.textViewMessageRowItemContactNumber);  
         this.rowItemMessageText = convertView.findViewById(R.id.textViewMessageRowItemMessageText);  
         this.rowItemSentStatus = convertView.findViewById(R.id.textViewMessageRowItemSentStatus);  
         this.rowItemSentTime = convertView.findViewById(R.id.textViewMessageRowItemSentTime);***
         1. ***Note:-*** Here we have used findViewById method, on the inflated view
      4. now set the data to be displayed in each text view  
         ***this.rowItemContactNumber.setText(currentMessage.getContactNumber());  
         this.rowItemMessageText.setText(currentMessage.getMessageText());***When setting text of above two text views, simply the values are taken from the message object ***this.rowItemSentStatus.setText(currentMessage.isSentStatus() ? "YES" : "NO");***
         1. When setting the send status, to the text view, we have converted it to a string values using a ternary operator in java

***this.rowItemSentTime.setText(dateFormat.format(currentMessage.getSentTime()));***

* + - 1. When setting the sent time, we have converted it to a string using the SimpleDateFormat object given below, which is defined in the class level  
         ***SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yy hh:mm a");***
  1. The complete method looks like below;



With this the implementation of the custom Array Adapter is completed

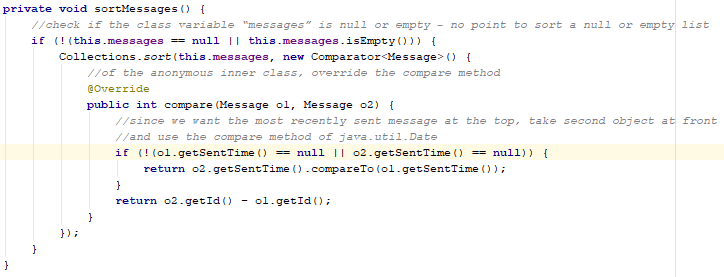
### Initializing List with messages from DB

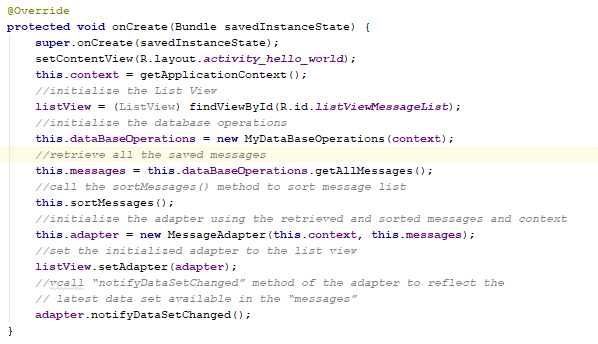
With the custom data adapter, parent view and row item view layouts ready, now its time to initialize the list view after retrieving messages from database. To recap, we have below components ready with us no;

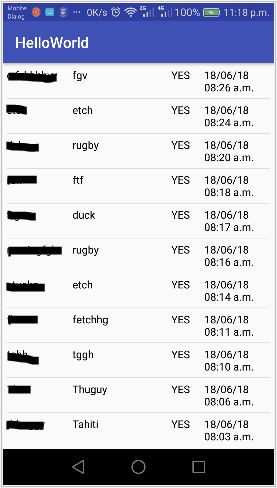
* **activity\_hello\_world.xml –** The parent layout containing the ListView
* **message\_row\_item.xml –** The layout to represent individual message inside the list view – a list view item representation
* **lk.uok.mit.adapter.MessageAdapter** – The custom array adapter to bind values from database to the text views of our row item view
* **lk.uok.mit.database.MyDataBaseOperations** – To retrieve all the messages from database

Using above components, let’s see how we can display the messages in the SQLite database inside a list

1. If you could recall, we used the “**lk.uok.mit.helloworld.HelloWorldActivity**” previously to demonstrate how to pass data from one intent to another. Since we are reusing it to demonstrate the ListView with a custom adapter, previous code is no longer required.
   1. Remove all the previous code from inside “**onCreate**” method except for below three lines  
      ***super.onCreate(savedInstanceState);  
      setContentView(R.layout.activity\_hello\_world);  
      this.context = getApplicationContext();***
   2. Create below class level variables;  
      ***//a variable to hold the message list retrieved from database  
      private List<Message> messages;  
      // a variable to hold the list view view  
      private ListView listView;  
      // a variable to hold the adapter  
      private MessageAdapter adapter;  
      // a variable to hold the database operations  
      private MyDataBaseOperations dataBaseOperations;***
   3. Rather than simply displaying the message list in the order we fetch them, lets write a method to sort them, so the most recently sent message will be listed at top  
      Write a method to sort the message list;
      1. Method Name :- **sortMessages**()
      2. Method Return Type:- void
      3. Method Parameters:- none
      4. Method Access Modifier:- private  
         Method Logic: -
         1. check if the class variable “messages” is null or empty – no point to sort a null or empty list  
             ***if (!(this.messages == null || this.messages.isEmpty())) {***
         2. Using the **sort()** method of **java.util.Collections** class, sort the messages – pass the message (list to be sorted) and an anonymous inner class of type “**java.util.Comparator**” to the method, and override the compare method as shown below  
            ***Collections.sort(this.messages, new Comparator<Message>() {  
             @Override  
             public int compare(Message o1, Message o2) {  
             if (!(o1.getSentTime() == null || o2.getSentTime() == null)) {  
             return o2.getSentTime().compareTo(o1.getSentTime());  
             }  
             return o2.getId() - o1.getId();  
             }  
             });***
      5. The complete method will look like below;



* 1. Inside onCreate method
     1. initialize the List View  
        ***listView = (ListView) findViewById(R.id.listViewMessageList);***
     2. initialize the database operations  
        ***this.dataBaseOperations = new MyDataBaseOperations(context);***
     3. retrieve all the saved messages from database and assign to “***messages***”  
        ***this.messages = this.dataBaseOperations.getAllMessages();***
     4. call the **sortMessages()** method to sort message list  
        ***this.sortMessages();***
     5. initialize the adapter using the retrieved and sorted messages and context  
        ***this.adapter = new MessageAdapter(this.context, this.messages);***
     6. set the initialized adapter to the list view  
        ***listView.setAdapter(adapter);***
     7. call “***notifyDataSetChanged***” method of the adapter to reflect the latest data set available in the “messages”  
        ***adapter.notifyDataSetChanged();***
     8. Now the complete “**onCreate**” method of **lk.uok.mit.helloworld.HelloWorldActivity** looks like below;  
        

1. With this the data retrieval and display part is completed, lets run the app and check  
     
     
   Source code for this tutorial part can be found in Git Repository given below: - <https://github.com/nadee158/android_tutorial_part_4.git>

# References

*Android AutoCompleteTextView Example*. (2018, January 1). Retrieved from JavaTPoint: https://www.javatpoint.com/android-autocompletetextview-example

*Application Fundamentals*. (2018, May 2018). Retrieved from Android Developers: https://developer.android.com

CHUGH, A. (2018, April 2). *Android AutoCompleteTextView Example Tutorial*. Retrieved from JournalDev: https://www.journaldev.com/9574/android-autocompletetextview-example-tutorial

Kapil. (2016, March 21). *Android SQLite Database Tutorial – CRUD Operations*. Retrieved from androidtutorialpoint: https://www.androidtutorialpoint.com/storage/android-sqlite-database-tutorial/

MULLIS, A. (2017, November 11). *Android Studio tutorial for beginners*. Retrieved from Android Authority: https://www.androidauthority.com/android-studio-tutorial-beginners-637572/

OGBO, O. (2016, September 21). *How to store data locally in an Android app*. Retrieved from Android Authority: https://www.androidauthority.com/how-to-store-data-locally-in-android-app-717190/

SINICKI, A. (2017, December 16). *Android SDK tutorial for beginners*. Retrieved from Android Authority: https://www.androidauthority.com/android-sdk-tutorial-beginners-634376/

Vogel, L. (2016, June 20). *Getting started with Android development - Tutorial*. Retrieved from vogella: http://www.vogella.com/tutorials/Android/article.html#high-level-overview-of-android-development

*What Is SQLite*. (2018, JUNE 01). Retrieved from Sqlite Tutorial: http://www.sqlitetutorial.net/what-is-sqlite/