TO-DO LIST APP - DOCUMENTATION

Introduction

It was agreed upon that the original four members of the group working on the TO-DO list would continue working on it and would recruit two new members. This was done by Thursday the 26th of November. This aim of this documentation is to provide a comprehensive understanding of what was undertaken in the four weeks of the project.

Group Members:

Sid Gupta
Sarah Kingston
Diarmuid McDonnell
Isla Hoe
Edmond O Floinn
Eoin O'Gorman

Initial Meetings

- Group fully formed

- Agreed on short meetings 3 times a week. This was due to a lack of meetings in the first 8 weeks and the idea that a few short meetings would result in a larger amount of productivity.

- The main six tasks were assigned as follows.

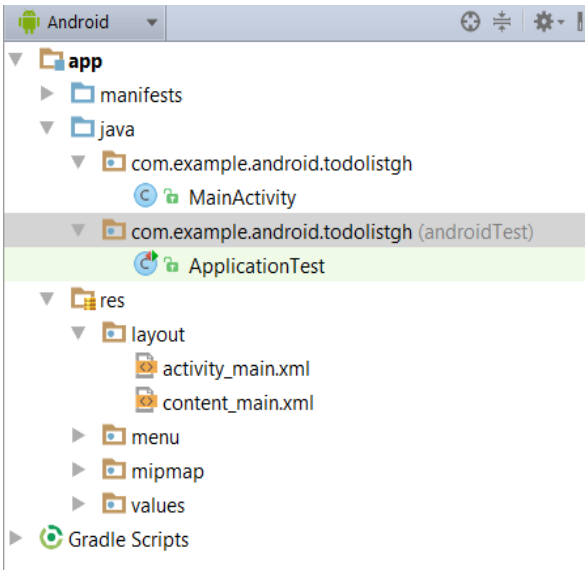
Functionality	*(2 main, 4 minor)*	*Edmond, Sid*
Appearance and Interaction	*(1 main)*	*Sarah*
Testing	*(1 main)*	*Diarmuid*
Documentation	*(1 main)*	*Isla*
Management and Collaboration	*(1 main)*	*Eoin*

- Using the previous code Edd started to separate it into classes to make it easier to read and to edit for each person.

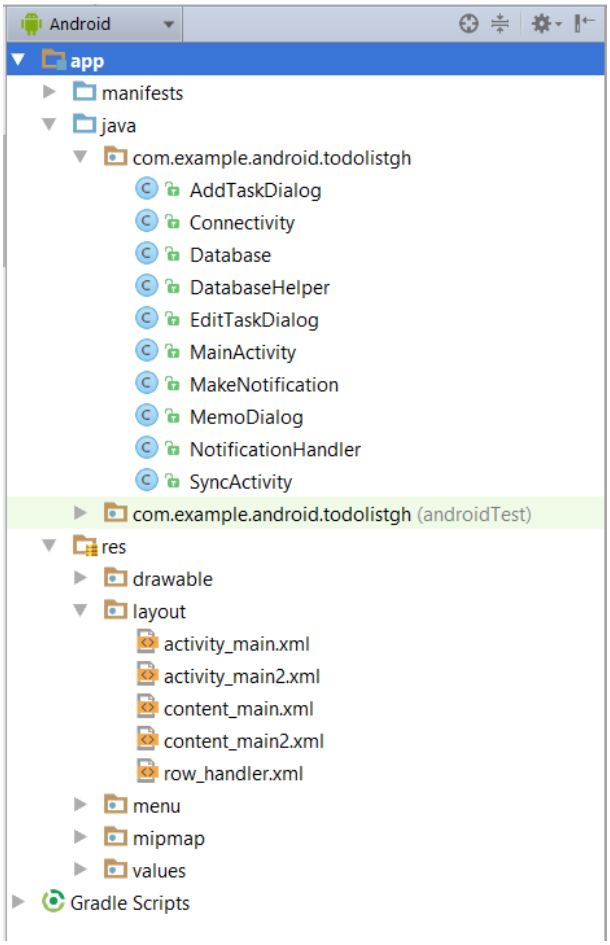
- A number of app features were listed for implemented if possible:

    - Save state (To include, what the task is, due date, location, has it been completed, location, category)
    - Store date and time
    - Push notification
    - Integrate with web?
    - Login
    - User table, username/password combo
    - Password encrypted

Code Structure



The above screenshot shows the original app code structure. As shown here under the "Layout" tab is only one activity_main and content_main and under the "Java" tab there is only one MainActvity. Due this the starting code was quite jumbled and not very easy to edit if the writer was not the original writer. Because of this the first task was to clean up the structure of the master in android studios. This was done by separating different elements of the code in to different classes.



The above screenshot shows how the classes were separated. This was the main editing to the code that was implemented in week one.

As well as these changes a number of tasks were assigned to separate people in accordance with the original plans for the app. These were as follows. The databases were to be primarily handled by Edd, the database is how the app stores information that is manually input by the user such as the date and the title of the task or a memo. The formatting of the date was to be handled by Sid, which is to make it easy to enter the date and to ensure that the date hasn't already passed, for example if it is December 4th 2015 a date before that such as December 1st can not be entered. The XML code which is used for the user interface and the appearance of the app was assigned to Isla, with Sid to also work on it. Priority of the tasks inputted by the user was addressed by Eoin and in conjunction with Isla, Sarah worked on coming up with a number of layout designs for the app using Draw.io.

The main functions are explained below with explanation of the code and how it works.

Functionality Explained

AddTaskDialog:

Dialogs in Android must be triggered by some way. One way to implement this is to use a thing called an interface, which is widely used in Java. An interface is used when a listener is set (onClickListener is a widely used interface). An interface cannot define a method, it can only say what should be called at a given time. It can be used in a class by using the keyword "implements" beside the class name.
In our app, we used an interface to create a custom dialog box. This is triggered on multiple occasions:

1. When a button has been pressed
2. When a long press has been detected

When the add button gets pressed, the interface for the addDialog class gets implemented. This calls the listener to set the button as a way of launching the dialog box. The field to be shown were written programmatically in Java instead of XML, as this was an easier way of achieving this. When the user clicks done, the fields are acquired, stored as variables, and if they are not null; they are given their own ID and row in the SQL database.

Similarly, other dialog boxes are triggered when the user long clicks on a record. This opens another dialog box that allows the used to either delete the record from the table, or modify it. If they delete it, the row is removed; the ID is deleted and the rest of the row's columns is purged.

Connectivity:

The purpose of the Connectivity class is to provide the Email Notes functionality.

We first need a default constructor- this constructor is used whenever an object of the "Connectivity" class is called without providing any parameters or information (for example if you say *"int x;"* and don't give x a value to begin with, x is automatically set to 0. This default constructor does the equivalent.

Next we have the actual function which launches the user's default email app and creates the email. First it must check if something has been entered. If something hasn't, the program shouldn't create the email. Instead it should inform the user that the memo is empty with a toast.

If the memo field is filled with some text, it's okay to email. So we must do the following:

1. Create an intent. An intent allows you to launch a new activity.
2. Specify that this intent will launch an email app (this is what Uri.*parse*("mailto:") does).
3. Then we must provide information on how to populate the email, namely what information will be filled into the following fields:

    a. the subject of the email
    b. the text in the body of the email

4. Lastly we must check that the user actually has an email app installed! If they do, go ahead and launch the email app. Otherwise we must not carry out the launching of the email app, as this will cause our app to break.

Database:

The database class keeps track of the column names in the SQL database. All columns in the table are stored in the string format. Each table has a table name and a row for IDs. IDs make it possible to track what row the columns are on to be accessed.
This class prevents the programmer from having to refer to column names in quotation marks. For example, instead of writing "category" as the column to be accessed; the programmer can write Database.TaskTable.CATEGORY instead.
The strings are declared as being "final" and "static" as they are not changeable, and are static, which infers that we do not want to make an object of the class to just access the variables.

Instead of writing:
Database database = new Database();
String column = database.TaskTable.CATEGORY;

The programmer can simply write:
String column = Database.TaskTable.CATEGORY;

Which saves on memory usage and makes things quicker to access. Final or static variables are conventionally written in block capital letters as a nomenclature.

Database Helper:

The DatabaseHelper class allows the programmer to add, modify, remove and query the database. Methods in here are generic, which means that given any table name, and specific parameters required; it's possible to edit anything within the table. The more generic a method is, the better it is; as it means it can be used in a variety of contexts without having to rewrite or create more methods to perform something similar with different parameters.

Queries are commands written in SQL, and are also final and static. These allow the correct rows to be chosen, or correct actions to be performed on the table itself.
For instance, CREATE_TABLE_QUERY is the query called when the table doesn't exist, and has to be created on the first run of the app. This is called in the method onCreate(). It creates a table and identifies the correct types to assign to each column. The ID per row is an integer primary key, which means that each row is assigned a unique, positive, real number which doesn't occur more than once per table. Category, task, due_date, and raw_due_date take the forms of text, as they are just holding strings. due_date is the field that shows the selected calendar date in the given locale of the phone (dd/mm/yyyy in a UK locale, mm/dd/yyyy in a US locale for instance), and raw_due_date is the date actually used for sorting in the SQL table query if the user wants to order the items in the list by a certain order (for example, due date ascending/descending, alphabetically ascending/descending, etc.).

Each database should have an assigned integer as its version. When the app is released, and thereafter the database is updated so that is has more or fewer columns, the data already in the database mightn't fully correspond to what is going to be added in the future (as the rows already there won't have any data in those added fields). Therefore it's important to migrate the data already in the table fully over and ensure it all works correctly. onUpgrade() is called when this happens. It takes the old database version, searches for data that is already in the database, and moves it over with its new fields initialised to null, so that records can be modified and correctly used.

A cursor is what is used to traverse a database. The cursor needs a query to search for the appropriate data. The methods for editing records need to know the ID, the row and col to be modified, and the appropriate data. The cursor, from MainActivity, gathers this information and checks each individual row to see if the IDs correspond.

EditTaskDialog:
EditTaskDialog works on the same principles as addTaskDialog, except the data from the database has be passed to the text fields first, so that the used can see what they have already entered, and edit accordingly. This dialog calls the editTask() function from the database helper, where the data passed as parameters are used as the new values for the given record with the id as a parameter in order to locate the record in the table.

MainActivity:

- Global declarations
- onCreate

The onCreate function is called once that activity has been started (in this case, when the app is launched). We first specify that we want the app to reload completely, and not just that particular activity in that instance. This is done by:

```
super.onCreate(savedInstanceState);
```

We then go on the specify that this activity should be formatted as specified in the *"activity_main.xml"* file as follows:

```
setContentView(R.layout.activity_main);
```

We must then also check to see how the list was ordered the last time the app was run (for example the user could have ordered it by descending date, and would hence want to see it ordered like this when they return to the app). It retrieves information on this ordering format, which we then use later on to format how the tasks are initially laid out. This is done by the sharedPreferences code:

```
sharedPreferences = PreferenceManager.getDefaultSharedPreferences(this);
editor = sharedPreferences.edit();
ordering =
sharedPreferences.getInt(getResources().getString(R.string.pref_key_ordering), 8);
```

Note that the value "8" is a default ordering value (e.g. "0" would be order by descending date, "1" is order by ascending date).

We must then declare each view that is used in *"MainActivity"*, and link these declared views to their XML counterparts. This is done using *"findVewById"* as follows (note that if they don't have an XML counterpart, the keyword new is used):

```
clearButton = (TextView) findViewById(R.id.clearAllTasks);
tableLayout = (TableLayout) findViewById(R.id.list_table);
totalCheckBox = (CheckBox) findViewById(R.id.select_all);
actionAdd = (com.getbase.floatingactionbutton.FloatingActionButton)
findViewById(R.id.action_a);
actionMemo = (com.getbase.floatingactionbutton.FloatingActionButton)
findViewById(R.id.action_b);
orderByDate = (TextView) findViewById(R.id.dateTitle);
orderByCategory = (TextView) findViewById(R.id.categoryTitle);
orderByDescription = (TextView) findViewById(R.id.descriptionTitle);

checkBoxes = new ArrayList<>();
```

(note that this ArrayList of checkBoxes stores the state of all the individual checkBoxes…(1) checked…(2) non-checked...etc...)

```
databaseHelper = new DatabaseHelper(this);
```
(creates a new instance of databaseHelper. See dataHelper class description to know what this does)

```
populateTable(DatabaseHelper.SELECT_ALL_QUERY);
```
(populates table- see description of populateTable written below)

```
databaseHelper.printTableContents(Database.TasksTable.TABLE_NAME);
```
(prints table- see description of databaseHelper class)

```
orderTable();
```
(orders table to the way it was last ordered - see description of orderTable written below)

The onCreate function now lists a number of onClickListeners:

onClickListeners constantly check to see if something has been clicked, and if something has been clicked, execute the code in the Listener. These are the following onClickListeners within the onCreate function:

- totalCheckBox
- orderByDate
- orderByCategory
- orderByDescription
- clearButton
- actionAdd
- actionMemo

- onCreateOptionsMenu
    - This method is inherited from an Android activity, which places the three dots for an options menu on the toolbar by inflating an XML file of values in a certain order in the menu itself. This allows for settings and other activities to be reached, which remain unused in our app.

- onOptionsItemSelected
    - This method takes the XML placeholder text views from the options menu as explained above, and adds functionalities to each option selected. Each option in the menu is given an individual ID which can then be accessed through inspection. given the item selected in the action bar, the parent activity handles the clicks accordingly and passes the intent to the super class for the given item selected to be executed as appropriate.

- orderTable
    - This method takes the key set in the shared preference and returns the value of the key in the integer format. The key should be already set from if the user clicks on a field to order the table by some way, such as date, or alphabetically. Testing this key's value in a switch statement will show which order the table was set by through the user, and invalidate the layout accordingly so that the table is redrawn, and the table repopulated.

- populateTable
    - This method takes the parameter of a query. The query provided should be the order by which the table should be laid out. Using the DatabaseHelper class, populateTable() uses a cursor object to traverse the table, and return the data in each given field. For each new row in the SQL table, a row in the layout is created programmatically which the user can visually see. Each ID is set per row created, so that if the user long presses on a row to delete or modify, or checks an item as completed; it acts accordingly and performs the appropriate actions.
- getData
    - Java uses private variables with public set & get functions. As we want to ensure that data is not accessed or modified at any other given times, it is important to provide methods for accessing and changing this variable. getData() is a method that returns the values stored by setData(). If data has been set, it can be accessed appropriately by returning a value of the given type; in this case, a string.
- setData
    - setData() is a generic method which takes the parameters of the row and column to be accessed. The pair passed allows the cursor to move to the specific row, and then given that row; it accesses the data stored in the given column and stores it in the string format.
- strikeThrough
    - This method performs a bitwise operation. Text fields' states are held by bitwise operations through flags that have been set. There are flags for underlined, italic, bold, and many other types of text formatting set. In order to manipulate the given text, this method takes a parameter of whether or not the text is to appear with a strike-through appearance. If the boolean value is true for having been completed, the text on the given field is to have strike-through flags using an OR operator. Else if the value is false where there is not a completed field, the text's flag is inversely ANDed with the strike-through flags in order to return it to its normal state of regular, non-struck-through text.

MakeNotification:
This class contains the function which creates the notification. The function createNotification() takes in the pending intent from the NotificationHandler, the date the task is due, the description of the task, and the calendar that contains the current date and the time the notification is to be called (9am). If the date of the task matches the current date then the notification is pushed to the phone.

NotificationHandler:

This class contains the function showNotification(). This function's job is to run a task every day at 9am. This task is calling the createNotification() function. This is done for each task using a for loop in MainActivity.

SyncActivity:

This class is an independent activity that creates a set of fields where the user can send an email to the email of their choice. On click of the send button, the listener is awoken, and the text retrieved from text field is passed to the connectivity class, where its method is called to send an email to the parameters appended.

The meetings in the following two weeks mainly consisted of each person catching the others up to speed on the code they were working on and how it impacted the app.

A lot of the final layout and design was perfected in the last week at the same time as the presentation was being planned. Sarah have been keeping up with designs throughout the previous three weeks but it was in the final one that the designs were confirmed. The images below depict the initial design and the final designs.