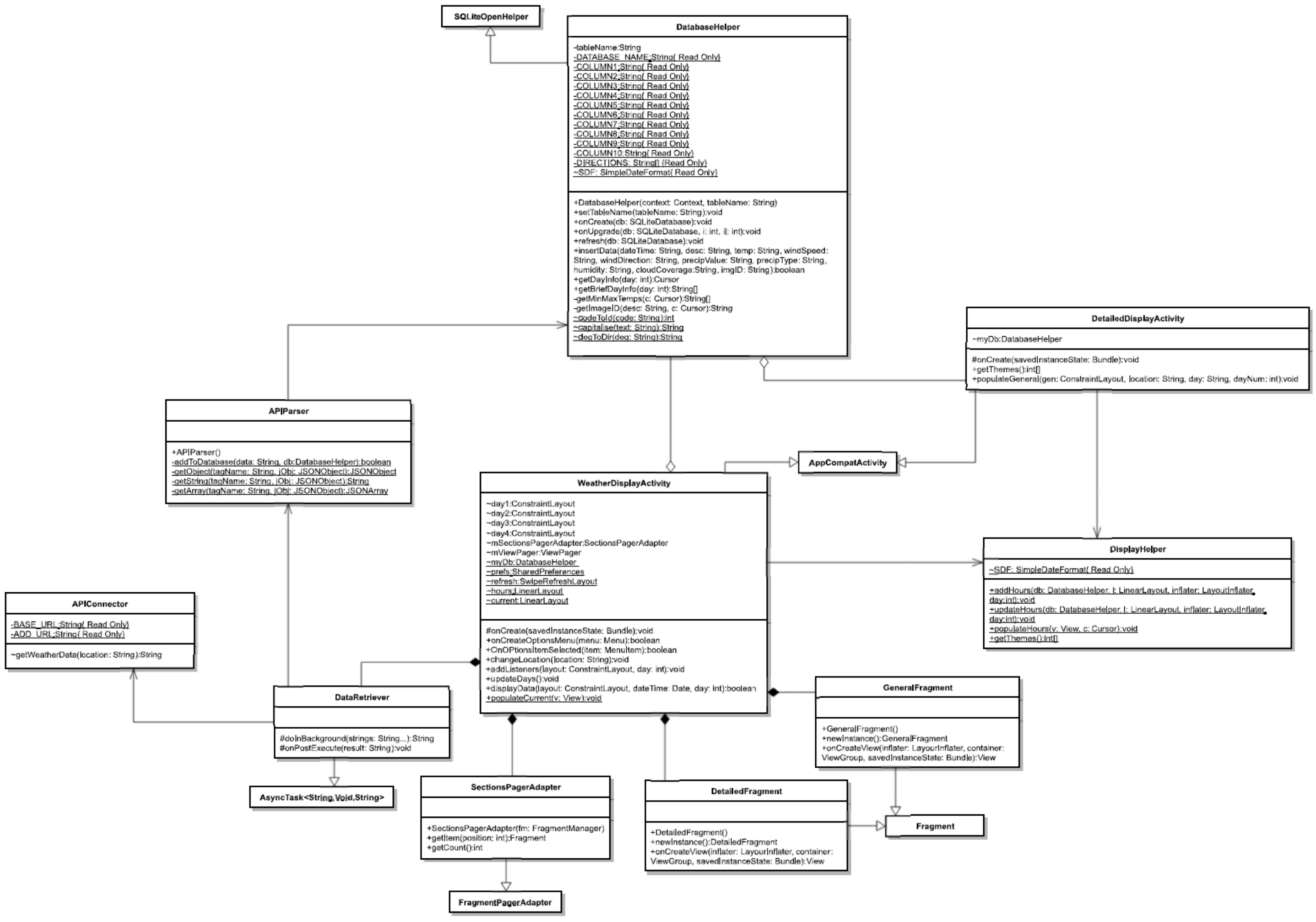
Report

UML Class Design

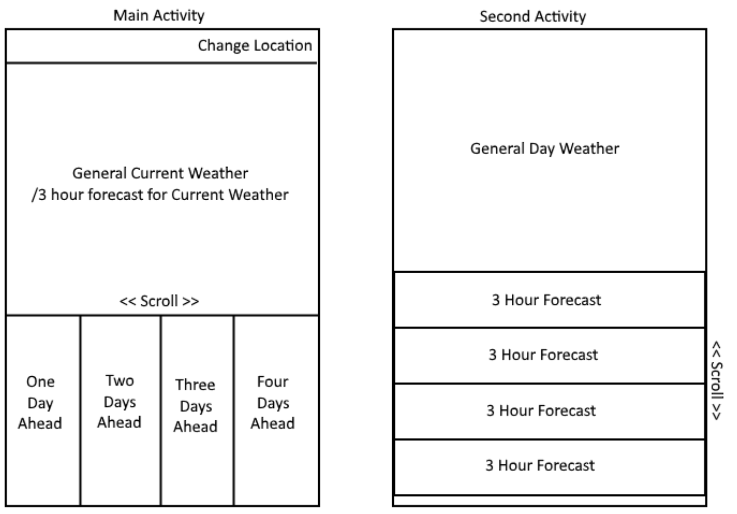

SQLiteOpenHelper
DatabaseHelper
-tableName:String
-DATABASE_NAME:String{ Read Only}
-COLUMN1:String{ Read Only}
-COLUMN2:String{ Read Only}
-COLUMN3:String{ Read Only}
-COLUMN4:String{ Read Only}
-COLUMN5:String{ Read Only}
-COLUMN6:String{ Read Only}
-COLUMN7:String{ Read Only}
-COLUMN8:String{ Read Only}
-COLUMN9:String{ Read Only}
-COLUMN10:String{ Read Only}
-DIRECTIONS: String[] {Read Only}
~SDF: SimpleDateFormat{ Read Only}
+DatabaseHelper(context: Context, tableName: String)
+setTableName(tableName: String):void
+onCreate(db: SQLiteDatabase):void
+onUpgrade(db: SQLiteDatabase, i: int, i|: int):void
+refresh(db: SQLiteDatabase):void
+insertData(dateTime: String, desc: String, temp: String, windSpeed: String, windDirection: String, precipValue: String, precipType: String, humidity: String, cloudCoverage:String, imgID: String):boolean
+getDayInfo(day: int):Cursor
+getBriefDayInfo(day: int):String[]
-getMinMaxTemps(c: Cursor):String[]
-getImageID(desc: String, c: Cursor):String
~codeToId(code: String):int
~capitalise(text: String):String
~degToDir(deg: String):String
DetailedDisplayActivity
~myDb:DatabaseHelper
#onCreate(savedInstanceState: Bundle):void
+getThemes():int[]
+populateGeneral(gen: ConstraintLayout, location: String, day: String, dayNum: int):void
APIParser
+APIParser()
-addToDatabase(data: String, db:DatabaseHelper):boolean
-getObject(tagName: String, jObj: JSONObject):JSONObject
-getString(tagName: String, jObj: JSONObject):String
-getArray(tagName: String, jObj: JSONObject):JSONArray
AppCompatActivity
WeatherDisplayActivity
~day1:ConstraintLayout
~day2:ConstraintLayout
~day3:ConstraintLayout
~day4:ConstraintLayout
~mSectionsPagerAdapter:SectionsPagerAdapter
~mViewPager:ViewPager
~myDb:DatabaseHelper
~prefs:SharedPreferences
~refresh:SwipeRefreshLayout
~hours:LinearLayout
~current:LinearLayout
#onCreate(savedInstanceState: Bundle):void
+onCreateOptionsMenu(menu: Menu):boolean
+OnOPtionsItemSelected(item: MenuItem):boolean
+changeLocation(location: String):void
+addListeners(layout: ConstraintLayout, day: int):void
+updateDays():void
+displayData(layout: ConstraintLayout, dateTime: Date, day: int):boolean
+populateCurrent(v: View):void
DisplayHelper
~SDF: SimpleDateFormat{ Read Only}
+addHours(db: DatabaseHelper, l: LinearLayout, inflater: LayoutInflater, day:int):void
+updateHours(db: DatabaseHelper, l: LinearLayout, inflater: LayoutInflater, day:int):void
+populateHours(v: View, c: Cursor):void
+getThemes():int[]
APIConnector
-BASE_URL:String{ Read Only}
-ADD_URL:String{ Read Only}
~getWeatherData(location: String):String
DataRetriever
#doInBackground(strings: String...):String
#onPostExecute(result: String):void
GeneralFragment
+GeneralFragment()
+newInstance():GeneralFragment
+onCreateView(inflater: LayourInflater, container: ViewGroup, savedInstanceState: Bundle):View
AsyncTask<String,Void,String>
SectionsPagerAdapter
+SectionsPagerAdapter(fm: FragmentManager)
+getItem(position: int):Fragment
+getCount():int
DetailedFragment
+DetailedFragment()
+newInstance():DetailedFragment
+onCreateView(inflater: LayourInflater, container: ViewGroup, savedInstanceState: Bundle):View
Fragment
FragmentPagerAdapter

Design & implementation

Layout

When designing the application, the main aim was to keep the application as simplistic as possible from a user perspective. Therefore, the application contains only two activities of which the user can navigate between.
The main activity displays the current weather in full detail as well as a brief overview of the weather for the next 4 days, in total displaying a 5-day forecast. The second activity gives a detailed forecast of any day selected by the user excluding the first, which the user can navigate to by ‘clicking’ the relevant day on the main activity. This was done to avoid having to use buttons in my application in an attempt to improve its overall look.
The displaying of a day’s forecast is broken down into two components, a general overview of the day and a detailed forecast at 3 hours intervals.
The general overview, as per my aim of simplicity, displays enough information to give the user a summary of what to expect from the weather on that day, containing the highest and lowest temperatures expected and the most common descriptor for that weather (i.e. if most descriptions

at 3 hours intervals say it is raining, the general description will say that it will be raining). It will also display an icon relevant to the common display to better illustrate it to the user.
The detailed forecast, at 3-hour intervals, displays more information to the user, including wind speed and direction, humidity, and precipitation information. Each interval appears as a rectangle in a list which the user can scroll through as they are within a vertical ScrollView. It also displays the time which the reading begins from and a smaller icon relevant to its description.
In the second display, the general display and detailed forecast (3 hours intervals) are displayed on top each other, however on the main activity this cannot be due to the space required to display all five days. As such, these two views are within fragments implemented by a PageView which allows for a user to navigate between the two displays by swiping left/right. The basic information for the next 4 days are displayed below.
The following is a crude design made early in the development of the application to illustrate these ideas.



Further, two themes have been created for the application to enhance the user experience. When the time is between 6AM and 8PM the bright ‘day theme’ is active, and outside those times the darker ‘night theme’ is active.
Finally, to ensure that the application works well on devices of any screen size and API level, no explicit sizes have been declared, meaning that they are proportional to the size of the screen, and, where possible, AppCompat versions have been used to ensure backwards compatibility amongst devices. The application is designed to run at API 26 but can run on an API as low as 15.

Storing Data
The user can change location by selecting a location from the options in the menu in the top right corner of the application. The current location is stored in the shared preferences of the app, allowing for that selection to persist between sessions. By default, the location is set to Exeter, and the user can also choose between Cardiff and London.
When data is returned and parsed from the API, it is stored in a single SQLite database in the relevant location table. This allows for the data to be read and the elements in the display be populated with the data. When the location is changed, it changes the table in the database in which the data is read from/stored in.
The application makes use of the ‘if exists’ keyword in SQL(ite) to ensure that a table for each location will always exist.
The application will attempt to update the data when it is initially loaded and can be explicitly updated by dragging down from the top of the screen. Using this gesture provides some familiarity

to the application as it is common amongst other mobile applications to link this gesture with this feature, making for an easier and more enjoyable user experience.

Reading Data
Data is always read from the relevant table to the current location and is read from the current date/time onwards, allowing for the data to appear that it is up-to-date even in the event when it isn't because of a failed update. If an update fails, then the user is informed through a 'toast' display. The program makes use of SQL query with dates to search through the database to return the data for each day and will therefore return no data for a date which the database has no data for.
The data is returned in a Cursor object and that is used to traverse the results from the aforementioned SQL query and process it for display. If no data is returned, due to an outdated database, then displayed instead are messages informing the user that no data exists and that they should attempt to update the database when possible. The application will also prevent the user from navigating to the second activity for that day if no data for it exists.

Code Structure
When writing the code, the aim was to ensure that the code itself was well encapsulated and to isolate any code that can be isolated. The DatabaseHelper class is solely responsible for the management of the database of the application and the tables within. It also contains methods regarding the retrieval and basic processing of the data from the database, such as extracting the maximum and minimum temperatures for a day etc. The call for data is made by the APIConnector class and it is responsible for connecting to the API and returning the data, if it can successfully make the connection. The data is returned as a JSON object and thus must be parsed so that its data can be inserted into the database. This parsing is handled by the APIParser class which uses JSONObjects and JSONArrays to extract the most important data from the API call.
Further, some methods are used by both activities and, as such, have been moved to a separate class, DisplayHelper. This class is mainly responsible for creating and populating the hour displays programmatically, although some other common methods are stored here. This is to promote code reuse and reduce any code duplication where possible.

Code Implementation
The general running of the application is as follows;

Main Activity
On creation of the main activity, the code first determines, via the use of a Calendar instance, which of the two themes should be implemented. This is determined first, before the layout is set, as otherwise the theme for this activity will not be set.
Once the layout has been set and allocated a background it acquires all references to any View in the display via the findViewById method and creates the two fragments and PageViewer for displaying the detailed weather for the current day. This is done through inflating the layout XML file programmatically.
Then the initial update method is called to ensure that data exists in the database before any methods attempt to access it which could result in exceptions, such as null pointers, due to a lack of data or a table.
In more detail, the updating is done in the DataRetriever class, a static inner class which extends AsyncTask, allowing it to make the API connection without compromising the usability of the application. It will first call a method of the APIConnector class which retrieves the data for the

current location from the API in a JSON format. The APIParser will then be called to extract each data item from the JSON data and insert each item into the database, via the DatabaseHelper.
Once the update completed, whether it be successful or unsuccessful, the relevant objects are populated with the data from the database. In the case of the hour displays, they are created via inflating the hour layout XML file in a loop, which runs a number of times equal to the amount of entries in the database query and populated with that data.
Finally, with regards to the future days (not the current day), they are populated with the data and, if data exists, an onClickListener is set which navigates to the second activity. If data does not exist then the onClickListsner is set to null, essentially disabling it.

Second Activity
On navigation from the main activity to the second, two extra values are passed through via the intent. The day name (Monday, Wednesday etc) and day number (1,2,3, or 4) are added as intent extras meaning that they can be extracted in the second activity.
Once the navigation has completed, a similar creation pattern occurs. Firstly, the themes are determined based off the system clock, although these themes differ slightly from the previous themes as they both hide the action bar, and all references for objects are obtained via the findViewById method. A new instance of a DatabaseHelper is created, however, since they both contain the same static final path, they both access the same data. Further, this activity does not have the ability to modify the data in the database as there is no updating possible. Finally, the extras passed with the intent are extracted and used to generate and populate the hours and the general information for the chosen day.

Changing Location
On the main activity the user can choose to change the location through the menu in the upper right of the screen. When the user selects a new value then the changeLocation method is called which edits the shared preferences, sets the table name in the DatabaseHelper instance and updates all information to match the data in the database. Note that, if the location selected is the same as the once currently active then the changeLocation method will not do anything.
Further, when updating the hours (detailed forecast display) it will compare the number of items to display against those that are already displayed and, if they differ, removes them all and re-creates them. If they match, then the values within them are overwritten.