**SWD504 Mobile Application Development - Report**

**How you have used XML layout files in your application**

Each activity has its own layout file in the layout folder in the project res. These files define the visual elements on screen, and the properties of these elements, such as the size and the ID that allows it to be identified in the code. The elements are laid out using a LinearLayout, or multiple of them nested together. The LinearLayout, as the name suggest, lays out elements in a horizontal or vertical line. This is not as useful on the main activity, as this only contains one element, the map, but is especially useful in the AddNewPOI activity. In the main activity xml file, there is just the code for the OSM map to fill the entire screen by matching the parent in both dimensions.

AddNewPOI’s layout file contains 3 TextViews, 3 EditTexts and a submit button. All of these components are inside a vertical layout. The first two textviews are within an additional layout, this one is instead horizontal in order to display the two elements side by side. They have their height set to wrap content so the element under them follows without overlapping. They have their width set to 0, but their weight set to 1 – this means they expand to fill the parent equally, as they are equally weighted – the weight is the ratio of all the elements, and allows the width to change dynamically depending on the device running the app. The two edittexts are also sized the same way within their own layout, so that the text appears as a label to the text entry field.

Back to the main layout, the textview, edittext and button are all displayed vertically. They have a height of ‘wrap\_content’ so they do not take up more room than necessary. The width is set to ‘match\_parent’ so that the text entry and button take up the entire width of the screen as they are the only element on that horizontal line.

The list xml file appears to only have 2 textviews displayed vertically, filling the width of the screen and setting the height to the content height, which is different here as one text is larger than the other. These 2 elements actually make up each entry in the list in the ViewPOIList activity.

**Use of findViewById()**

Although the elements are defined within the XML file, they must be linked to manually in the Java class. We do this using findViewById(). A view is an basic object that represents a basic building block for the visual interface. These can be identified using the id that they are assigned in the XML layout. When we store the retrieved view, we have to specify the type – for example button, EditText and MapView. Once these classes and instantiated as objects, we can manipulate them using functions. For example to get the value out of an edittext we use id.getText().toString();.

**Communication between activities using Intents and Bundles, and result codes where appropriate**

In my code there are 3 activities in addition to the main activity. To launch these, an intent is created with the current context and the name of the activity to be launched. An intent is a class that android uses to navigate between activities. The activity can then be launched with startActivity(intent). The ‘AddNewPOI’ and ‘ViewPOIList’ activities both send back data, so instead are launched startActivityForResult(intent,[request code]). The request code is so that when the main activity is returned to, the activity that it has returned from can be identified and data sent back can be processed correctly.

A bundle is an object that allows objects to be stored with a key mapped and passed between activities. The key can then be used to retrieve the data in the destination activity. Once the bundle is instanced, data can be stored using bundle.put[type](data);. For example, in the adding a new poi screen, the name, type and description are all stored in the bundle using bundle.putString([data]);. The bundle is then stored in the intent using intent.putExtras(bundle);. We have to specify a result code for when the app changes back to the main activity, in both cases as RESULT\_OK to show a success. This is combined with the bundle and the finish(); function uses the intent to go back to the main activity.

Back in the main program, there is an onActivityResult function. Within this are if statements to check for the various requestcodes. Following a branch and once passing an if statement to check the RESULT\_OK, the bundle is retrieved out of the intent with intent.getExtras() and values retrieved with extras.getString[key];.

**Use of preferences**

The actual preferences and their type are defined within the preferences.xml file in the res folder. In my app there is only one preference of a checkbox type for auto-uploading POIs. The preference menu is accessed from launching the preference activity from the burger menu. This activity is declared as extending the built-in PreferenceActivity, and the onCreate uses R to link to the preferences.xml file. Back in the main program, the preferences are retrieved by using a PreferenceManager to get the preferences for the whole app. Individual preferences are then retrieved using prefs.getBoolean(); in this case for getting the autoupload preferences.

**File I/O**

Saving to file is handled by a static method within the POIList class, which is called whenever this is needed. This works by instancing a PrintWriter object, which in turn takes a new FileWriter object as a parameter, using ‘Environment.getExternalStorageDirectory().getAbsolutePath()’ to navigate to the device-specific file directory, and then appending the file name on the end. A for loop is then created with the contents of the POI list as a condition. This is a list of POI objects, a class I created to hold the details for each POI. For each iteration, getters are used to retrieve the information and it is written out line by line. This is then closed at the end when all the POIs are in the file. The entire saving process is wrapped in a try-catch to prevent file handling errors. The program returns true or false depending on the result of the function, which is used to give the user feedback in the main program.

Loading from file is similarly handled by a static method in the POIList class. A FileReader is created for the device’s file directory of the same name as for saving. This time a buffered reader is created to read the file line by line. A while loop is used to read each line until the end of the file, parsing the CSV (described later) and retrieving the 5 components before making a new POI object and storing it into the list. Once again this is wrapped in a try-catch, as if the user tries to load the files before saving any POIs, the app would crash otherwise. Instead, this returns false rather than true if it has succeeded, showing a relevant message to the user in the main program.

**Network communication, including both GET and POST requests as appropriate**

For both uploading and downloading an AsyncTask was created, as these functions take a long time to execute and would normally be killed off as unresponsive if they were in the main thread. AsyncTask runs these in a separate thread to prevent this.

To download the waypoints, a class DownloadTask that extended AsyncTask with no parameters was created. Within this class’s doInBackground method, a try-catch statement was written to contain the code. A new HTTP connection object was created, as well a URL object for the GET request that returns the POIs in csv format. The URL is passed into the connection and an input stream opened with the connection. If the response code is 200, meaning OK, this is then passed into a BufferedReader. The POIs are then parsed from the csv using the same code as loading from a file, and loaded into objects. The POIs are all redrawn (with some extra code to get around removing the ‘you are here’ marker) onto the overlay from the POIList in a for loop. The function then returns a positive message to the user. If the response code is not 200, indicating an error, this is instead returned. The returned values go to the onPostExecute() function within the AsyncTask, where they are put into an AlertDialog to the user.

The UploadTask AsyncTask takes the 5 items to make a POI as a parameter. Much like downloading, a HttpURLConnection and URL created and opened together. As this is a POST request, rather than appending query strings on the url, the post data is built in a string using the input to the AsyncTask. These inputs must be encoded to UTF-8 to prevent characters such as spaces breaking the URL and causing errors. This post data string was passed into an OutputStream and posted out. If the response code is 200, again meaning OK, a BufferedReader is opened to read back the response echoed out by the script, and return it to the onPostExecute. Again, this is all within a try catch to catch any file errors and return the error to onPostExecute. onPostExecute displays a toast message if the upload was a success, or shows an alert with the server response to the user.

An important part of both AsyncTasks is the inclusion of a ‘finally’ at the end to close the connection to prevent memory leaks if the task does not finish properly.

g) JSON or CSV parsing, as appropriate