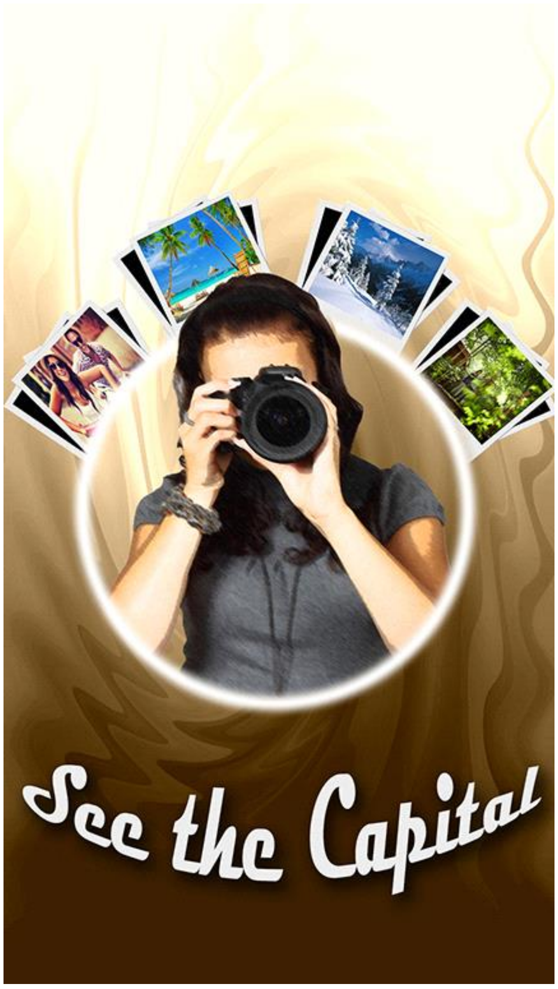
CS385 Mobile App Development Project

SeeTheCapital Android Application



Team name:

The Militant Fireballs

Team members:

Sinead O'Rourke

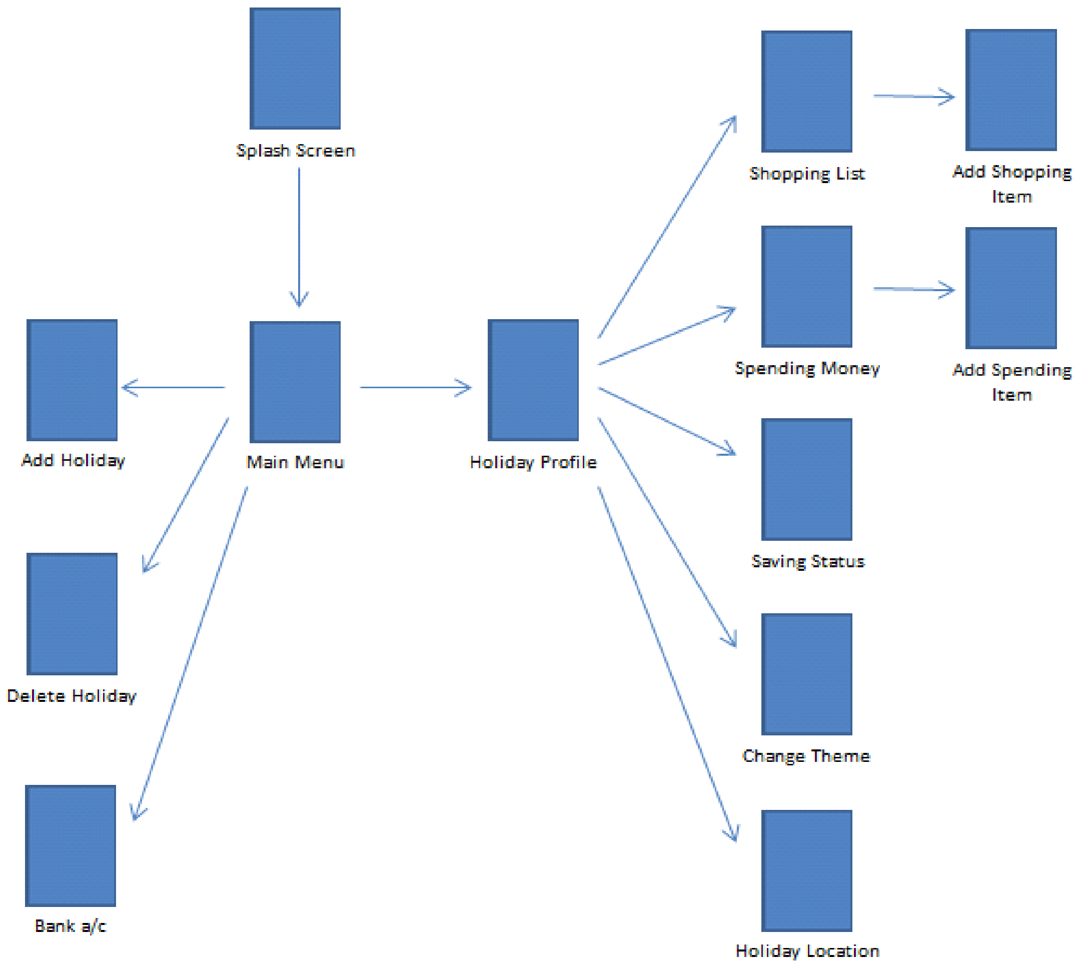
Tommy Coyne

Outline

SeeTheCapital is an app to help the user plan their personal savings before an intended holiday. Customers can input details about travel price, what they need to buy before departing and what they need to budget for while away, and the app returns a breakdown of how much still needs to be saved before the departure date. The app makes the budgeting experience fun and personal by allowing the user to define aspects of the design and by sending the holiday-goer encouraging messages to keep saving. *SeeTheCapital* is targeted at a broad range of clients, whether they are:

- a travel enthusiast or first time holiday maker.
- an organized person or someone who needs to start keeping a closer eye on their spending.
- a student with their first savings account going inter-railing or a more mature client planning their retirement getaway (not to forget everyone in between).
- any gender, any nationality.

Screen Flow



Methodology

The aim was to have a user-friendly layout that not only looks good, but is also easy to follow. After the splash screen, the customer is brought to the overall Main Menu with buttons linking to other activities (to access the database to add or delete an intended holiday, or to access the simulated bank account to add or subtract money). After a holiday is added to the database it is displayed in a ListView in the main menu. Here the user can click on a destination and the app brings them to the Holiday Profile for the selected destination. The Holiday Profile is set up with a Navigation Drawer with easy access to the following options:

- add/delete an item from the selected holiday's Shopping List
- add/delete an item from the selected holiday's Spending Money budget
- view a breakdown of the selected holiday's saving status
- change the background design of the selected holiday's profile
- browse the location of the selected holiday with google maps

As a team the project tasks were split out among each member into two workloads:

- the database, simulated bank a/c, savings status
- designs, change theme, holiday location

The team used a combination of Dropbox and email to communicate, and met up three times outside of lab hours. The team used a number of lecture notes from Moodle and online resources to achieve the production of *SeeTheCapital*.

Splash Screen

To create the splash screen, there are three files that needed to be either created or added to for the splash screen to work correctly.

The android manifest file needed to be adjusted with the following code.

```
<activity android:name=".SplashActivity" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

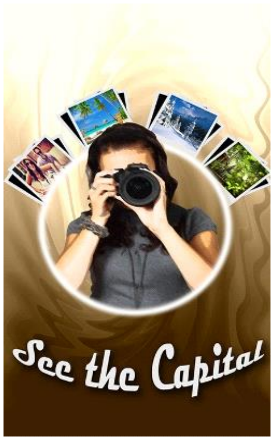
The splash screen is the launcher activity of the app therefore the first activity to run when the app is opened.

The rest of the activities are default, but are embedded in an intent filter, which means that you could move to the various activities when a button is pressed etc.

We then create the splash activity xml file which controls the layout of the activity when the app starts with the following code:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@mipmap/bg">

    <imageView android:id="@+id/SplashScreen"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layout_gravity="center"/>

</LinearLayout>
```

The Splash activity's background is set to load the main background image which is stored in various sizes within the mipmap folders which are found within the res folder.



The next step was to create the java class which included the import statements:

```
import android.app.Activity;
import android.content.Intent;
import android.media.MediaPlayer;
import android.os.Bundle;
```

The import statements also includes the media.MediaPlayer statement which is needed when calling the media player to play the sound at startup of app launch.

```
public class SplashActivity extends Activity {

    MediaPlayer splashSound;

    Thread timer = new Thread(){
        public void run(){
            try {
                sleep(6500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                Intent intent = new
Intent("com.example.sinead.seethecapital.MainMenu");
                startActivity(intent);
                finish();
            }
        }
    };
```

Under the Splash Activity, we create a Media Player object and if when it is placed outside of the main method it can be used globally.

The launched sound is set to a timer of 6500, which is 6.5 seconds of audio.

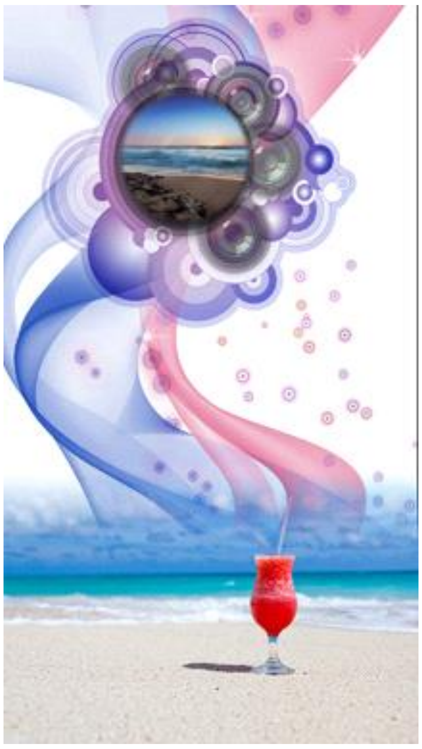
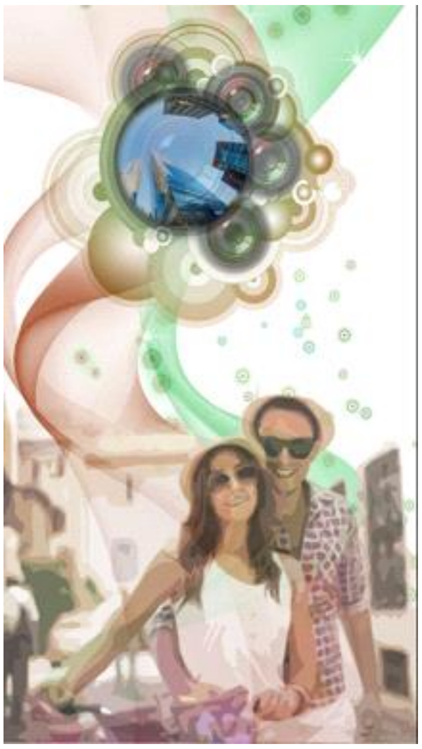
The audio then finishes once the timer ends and a new intent is started, which brings the user to the main menu of the app.

```
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_splash);
        timer.start();
        splashSound = MediaPlayer.create(this, R.raw.sound); /*Gets our sound-file
from res/raw/sound.ogg */
        splashSound.start(); //Starts our sound

//This is where we would continue with our run/thread-code.
    }
```

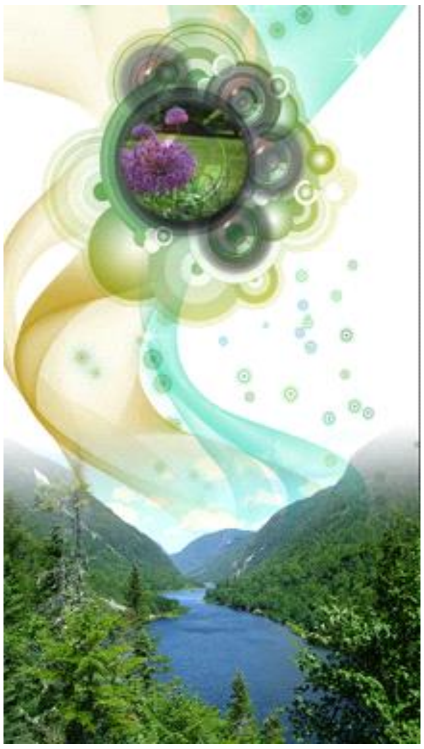
Within the onCreate Method we reference the audio clip which is called sound.ogg. The audio clip is stored in the raw folder and we call the audio clip to play with the mediaplayer.create method.

The audio clip ends with an onDestroy(); method being called once the splash screen enters in to the main menu activity of the app.

Themes

SUN HOLIDAY	CITY BREAK



COUNTRYSIDE	SNOW HOLIDAY

There are four different user defined dynamic themes available for the users to choose from when they want to personalise their travel itinerary.

All four themes, the main background image and the Launcher splash screen activity background were mainly designed using the Adobe Photoshop and illustrator design packages. The retro wavy lines are subtly shown in the splash background image on the splash screen and is then emphasised within the main holiday theme choices and these were created using the tools available within Adobe Illustrator.

The images then coincide with the individual theme choice by using the visual representation of a beach and an ice cold cocktail for the sun holiday theme. A snowy mountain and someone skiing to represent the snow break holiday and so on. The camera lens and their images are an indication to photographs that may have been or can be taken while on holiday.

Within the android manifest file we had to add a theme change activity.

```
<activity
    android:name=".HolidayChangeTheme"
    android:label="@string/title_activity_holiday_change_theme"
    android:theme="@style/AppTheme.NoActionBar" >
    <intent-filter>
        <action android:name="com.example.sinead.seethecapital.HolidayChangeTheme"
/>

        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

```
<style name="AppTheme.Beach" parent="Theme.AppCompat">
    <!-- alternative theme attributes -->
    ...
    <item name="themedSunDrawable">@mipmap/beach_main</item>
</style>
```

Each theme had to be given a unique style setting within the style.xml file and these would be then referenced within the content_holiday_change_theme.xml file when the user decided on their theme choice.

```
tools:showIn="@layout/app_bar_holiday_change_theme"
tools:context="com.example.sinead.seethecapital.HolidayChangeTheme"
android:id="@+id/changetheme">
```

The option of changing the theme is available from the activity_add_holiday.xml at the bottom of the screen and within that there is a spinner drop down menu where the user can choose their theme.

```
<Spinner
    android:layout_width="175dp"
    android:layout_height="wrap_content"
    android:id="@+id/theme_spinner"
    android:layout_below="@+id/returnCalendar"
    android:layout_alignLeft="@+id/submitButton"
    android:layout_alignStart="@+id/submitButton"
    android:layout_marginTop="30dp"
    android:background="#CCffffff"
    android:spinnerMode="dropdown" />
```

In order to tell Android to use different drawables for different app themes, we create custom attributes that allow specifying reference to the correct drawable, and provide different drawable references as values for these custom attributes under different themes (the same way appcompat library provides custom attributes such as colorPrimary).

When we have the polished themes ready to be used, we need to allow users to choose which one they prefer and switch theme dynamically during runtime. This can be done by having a Shared Preferences, says pref_dark_theme to store theme preference and use its value to decide which theme to apply. Application of theme should be done for all activities, before their views are created, so onCreate() is our only option to put the logic.

SplashActivity.java

```
public abstract class SplashActivity extends ActionBarActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        if (PreferenceManager.getDefaultSharedPreferences(this)
                .getBoolean("pref_main_bg_theme"), false)) {
            setTheme(R.style.AppTheme_MainBg);
        }
        super.onCreate(savedInstanceState);
    }
}
```

Here, since our app already has a default theme, we only needed to check if default preference has been overridden to override dark theme. The logic is put in the 'base' activity so it can be shared by all activities.

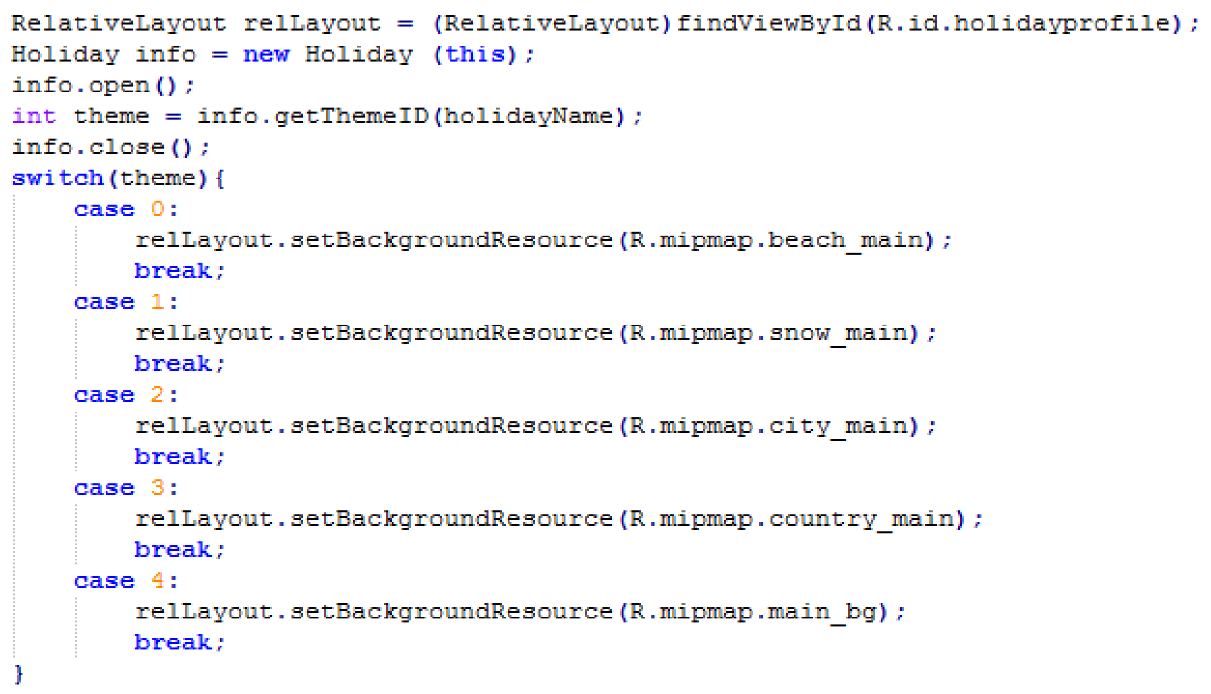
This approach only applies to theme activities that are not in the back stack. For those that are already in current stack, they will still exhibit previous theme, as going back will only trigger onResume(). Depends on product requirements, the implementation to handle these 'stale' screens can be as simple as clearing the back stack, or restarting every single activity in the back stack upon preference change. Here we simply clear back stack and restart current activity upon theme change.

SettingsFragment.java

```
@Override
public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        mMain = (MainActivity) getActivity();
        mMain.setTitle(getString(R.string.settings));
        updateSummaries();
        SharedPreferences spTheme = getActivity().getSharedPreferences("config",
Context.MODE_PRIVATE);
        spTheme.registerOnSharedPreferenceChangeListener(new
SharedPreferences.OnSharedPreferenceChangeListener() {
@Override
public void onSharedPreferenceChanged(SharedPreferences sharedPreferences, String
key) {
        if (key.equals("pref_main_bg_theme")) {
        mMain.recreate(); // <- calling getActivity().recreate() doesn't work for
this coding
        }
        }
        });
        }
```

Theme Changer

In each of the onCreate() methods of the activities in the holiday profile section of *SeeTheCapital* there is a switch case which will set the background of the activity according to the selected holiday's theme ID.



```java
RelativeLayout relLayout = (RelativeLayout)findViewById(R.id.holidayprofile);
Holiday info = new Holiday (this);
info.open();
int theme = info.getThemeID(holidayName);
info.close();
switch(theme){
    case 0:
        relLayout.setBackgroundResource(R.mipmap.beach_main);
        break;
    case 1:
        relLayout.setBackgroundResource(R.mipmap.snow_main);
        break;
    case 2:
        relLayout.setBackgroundResource(R.mipmap.city_main);
        break;
    case 3:
        relLayout.setBackgroundResource(R.mipmap.country_main);
        break;
    case 4:
        relLayout.setBackgroundResource(R.mipmap.main_bg);
        break;
}
```

The purpose of HolidayChangeTheme.java is to allow the user to change their mind about the background of the custom holiday profile section, and permanently update this decision in the database.

A SeekBar and an ImageView are declared in the java file and the .setOnSeekBarChangeListener() is implemented. In the override methods:

- onProgressChanged uses a switch case to change the ImageView as the slider is moved (acts as a preview of the background options)

- onStopTrackingTouch also uses a switch case to display a pop-up with the corresponding theme/background name when the user has stopped the slider on an image they like.

The new theme ID replaces the old theme ID in the database when the updateTheme() method is called.

Database

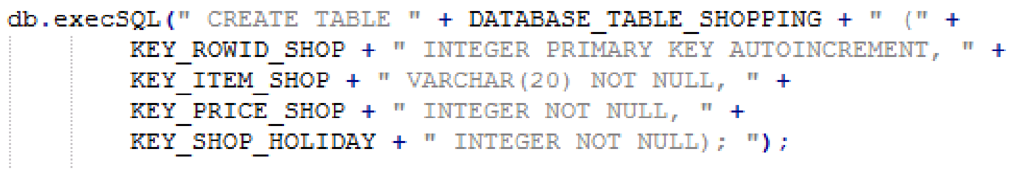
SeeTheCapital uses SQLite, which is an open-source relational database system. SQLite was chosen for this project as it does not require an external server to run. Instead, the files are locally stored on the mobile device and the information is permanently saved regardless of the app or mobile device being switched off. With the use of the DbHelper class, *SeeTheCapital* is able to execute SQL create, insert, update and delete statements to manage the database entries. All of the SQL implementation is done from the Holiday.java class.

Creating Tables

It was trial and error when it came to creating the right database for this app. Initially the idea was to have a holiday table, and then each row from this table would create its own shopping list and spending money table. This proved difficult to implement and would have used up unnecessary storage. For proper execution of all tasks, *SeeTheCapital* only needed to create four separate tables in the Holiday database.

- DATABASE_TABLE - to store the user defined holidays
- DATABASE_BANK - to store the simulated bank account total
- DATABASE_TABLE_SHOPPING - to store the user defined shopping items
- DATABASE_TABLE_SPENDING - to store the user defined spending money items

Below is an example of how the Shopping List table was created using the SQL execution statement.



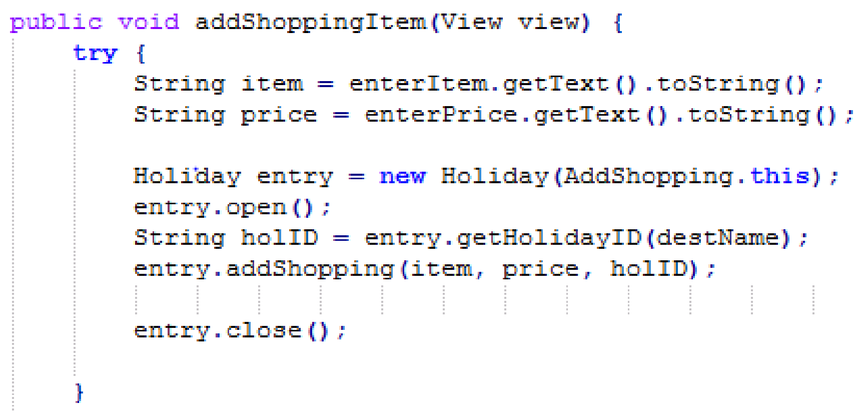
```
db.execSQL(" CREATE TABLE " + DATABASE_TABLE_SHOPPING + " (" +
        KEY_ROWID_SHOP + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
        KEY_ITEM_SHOP + " VARCHAR(20) NOT NULL, " +
        KEY_PRICE_SHOP + " INTEGER NOT NULL, " +
        KEY_SHOP_HOLIDAY + " INTEGER NOT NULL); ");
```

Here the Holiday.java class creates a table with four columns. The first attribute is a shopping item ID which is given to every shopping item that is logged in the database. This ID needs to be unique (as it is the Primary Key) therefore the database manages the creation of this ID itself (by using AutoIncrement). This also means that the ID does not need to be included in the insert statements, as it is automatically inserted.

The other columns include the shopping item name (which has a limit of only 20 characters long), the shopping item price (an integer) and a holiday ID (see below for explanation).

Although they do not have proper join tables connecting them, each shopping/spending item has to have the holiday ID of the destination they are referring to. This is done behind the scene (out of sight from the user) and is inserted into the shopping list/spending money table each time the customer adds a new item name and price.

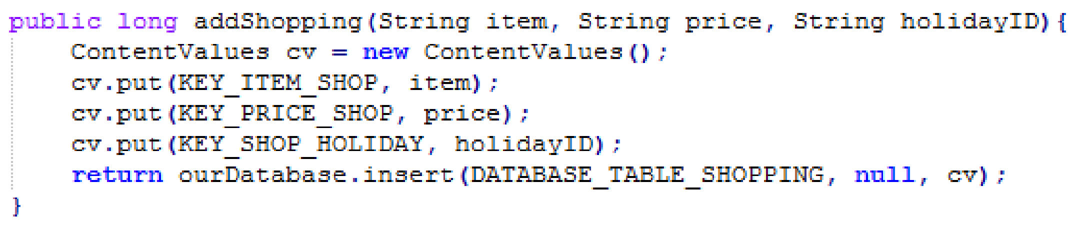
Below is a snippet of code to explain how a shopping item is inserted into the database. The method addShoppingItem() is called when the "save" button is clicked in the AddShopping.java file.



```
public void addShoppingItem(View view)  {
    try {
        String item = enterItem.getText().toString();
        String price = enterPrice.getText().toString();

        Holiday entry = new Holiday(AddShopping.this);
        entry.open();
        String holID = entry.getHolidayID(destName);
        entry.addShopping(item, price, holID);

        entry.close();

    }
```

The method first takes in a String of what the user typed into the item and price EditTexts. Next the method makes an instance of the Holiday.java class and opens the database. The getHolidayID() method is called by passing through the name of the selected holiday profile. This 'destName' is a class variable that was received from an intent (by using .putExtra() and .getExtras() methods) when the user was brought to the AddShopping.java.

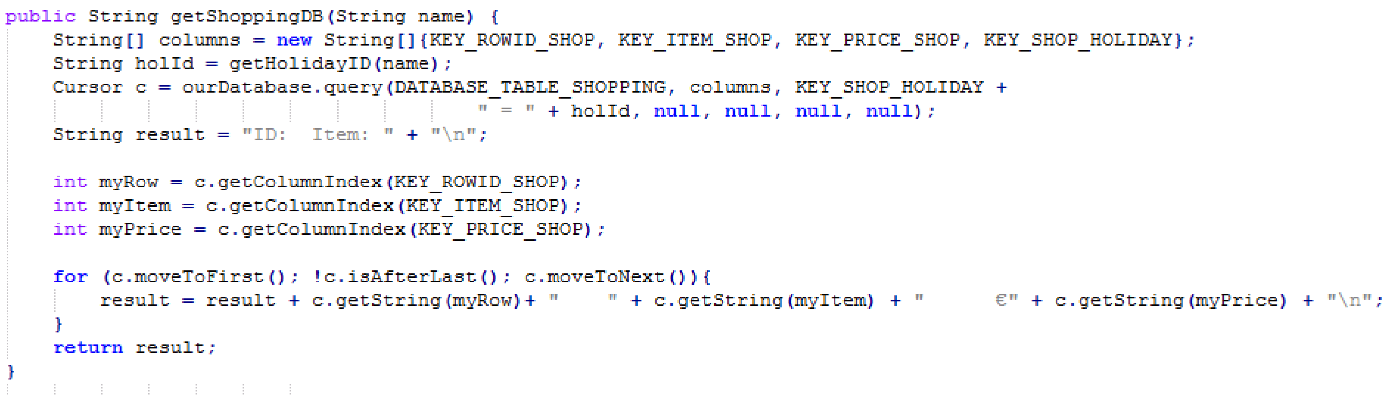
getHolidayID(destName) returns a String version of the selected holiday's ID, and along with the item name and price, the three Strings are inserted into the Shopping List table before the database is then closed. Below is an example of how the app uses .insert() to put the shopping item information into the appropriate table. The use of ContentValues allows the method to insert many columns with just one .insert() statement.



```
public long addShopping(String item, String price, String holidayID){
    ContentValues cv = new ContentValues();
    cv.put(KEY_ITEM_SHOP, item);
    cv.put(KEY_PRICE_SHOP, price);
    cv.put(KEY_SHOP_HOLIDAY, holidayID);
    return ourDatabase.insert(DATABASE_TABLE_SHOPPING, null, cv);
}
```

Queries

Now that each shopping item keeps record of the holiday ID, it is easy to display a unique Shopping List populated with items that correspond to that selected holiday profile only. A good example of this can be seen in the getShoppingDB() method in the Holiday.java file.



```
public String getShoppingDB(String name) {
    String[] columns = new String[]{KEY_ROWID_SHOP, KEY_ITEM_SHOP, KEY_PRICE_SHOP, KEY_SHOP_HOLIDAY};
    String holId = getHolidayID(name);
    Cursor c = ourDatabase.query(DATABASE_TABLE_SHOPPING, columns, KEY_SHOP_HOLIDAY +
                                    " = " + holId, null, null, null, null);
    String result = "ID:  Item: " + "\n";

    int myRow = c.getColumnIndex(KEY_ROWID_SHOP);
    int myItem = c.getColumnIndex(KEY_ITEM_SHOP);
    int myPrice = c.getColumnIndex(KEY_PRICE_SHOP);

    for (c.moveToFirst(); !c.isAfterLast(); c.moveToNext()){
        result = result + c.getString(myRow)+ "    " + c.getString(myItem) + "      €" + c.getString(myPrice) + "\n";
    }
    return result;
}
```

This method takes in the destination name of the selected holiday and after it gets the corresponding holiday ID to that destination, it uses a cursor to query the Shopping List table and return a result set of shopping items where their holiday ID equals the destination's ID. This is very similar to a SELECT statement in SQL.

Using a for loop, the cursor begins at the first row in the result set, saves this information to a String and then works its way down to the end of the result set (adding a row to the result String with each loop).

This String is then be displayed to the user by implementing a TextView and the .setText() method in the HolidayShoppingList.java activity.

Inserting

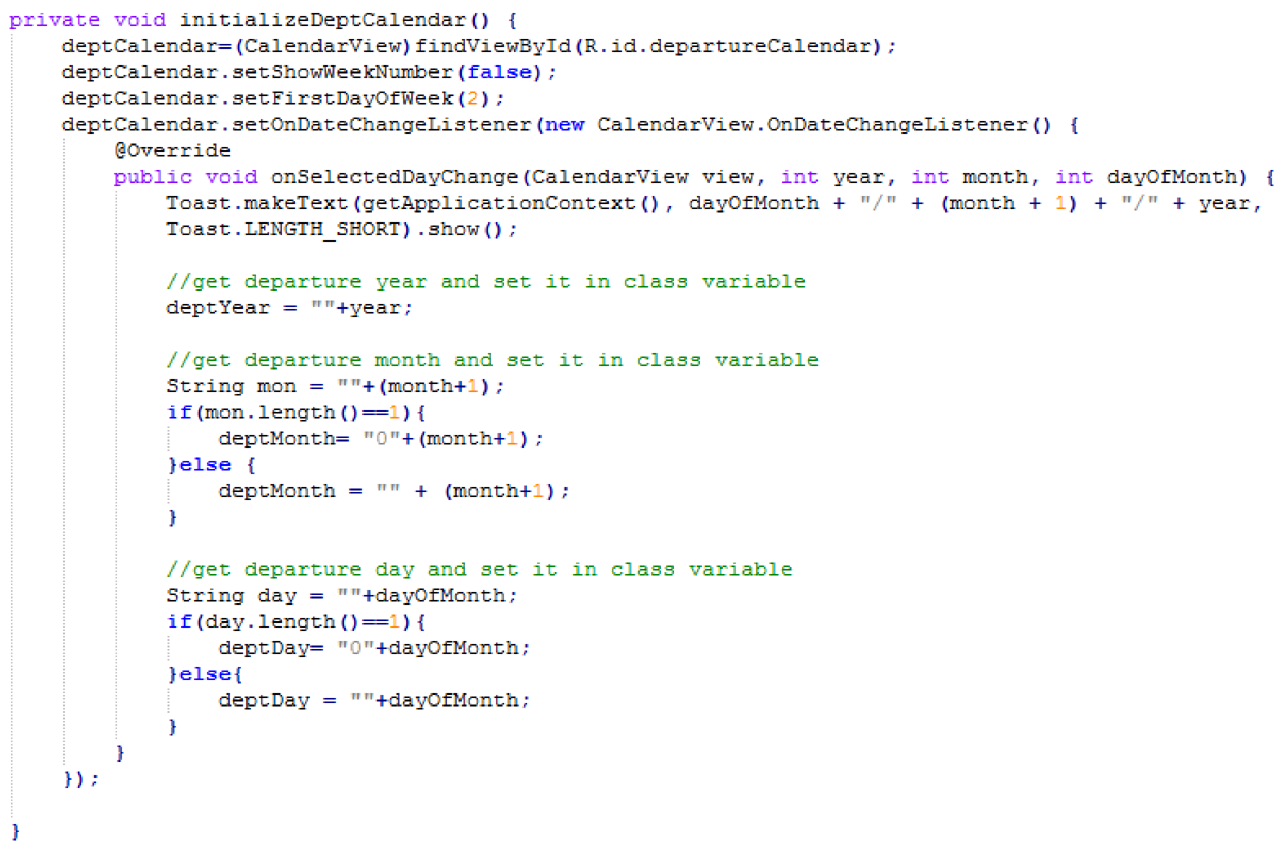
EditText boxes are not the only widgets used in *SeeTheCapital* to help the user input information into the database. CalendarViews and a Spinner are also used when information about a new holiday is being added (AddHolidayActivity.java).

<u>CalendarView:</u>

By selecting a date from either the departure or return calendars, the user is inserting that information into the Holiday table without having to manually type the day, month and year of the planned travel. Below is an example of how the app initializes the departure calendar and saves the selected date as a class variable.

First a new CalendarView is declared and references the xml file. The method sets the calendar to dismiss showing the week number (to save space on the screen) and sets the first day of the week to be Monday (=2).

When a date is selected by the user, a toast pop-up appears with that date, and the selected day, month and year are stored into individual class variables. This makes it easy to insert the date into the database at a later stage when the “save” button is clicked.



```java
private void initializeDeptCalendar() {
    deptCalendar=(CalendarView)findViewById(R.id.departureCalendar);
    deptCalendar.setShowWeekNumber(false);
    deptCalendar.setFirstDayOfWeek(2);
    deptCalendar.setOnDateChangeListener(new CalendarView.OnDateChangeListener() {
        @Override
        public void onSelectedDayChange(CalendarView view, int year, int month, int dayOfMonth) {
            Toast.makeText(getApplicationContext(), dayOfMonth + "/" + (month + 1) + "/" + year,
            Toast.LENGTH_SHORT).show();

            //get departure year and set it in class variable
            deptYear = ""+year;

            //get departure month and set it in class variable
            String mon = ""+(month+1);
            if(mon.length()==1){
                deptMonth= "0"+(month+1);
            }else {
                deptMonth = "" + (month+1);
            }

            //get departure day and set it in class variable
            String day = ""+dayOfMonth;
            if(day.length()==1){
                deptDay= "0"+dayOfMonth;
            }else{
                deptDay = ""+dayOfMonth;
            }
        }
    });

}
```

One small issue did arise with the CalendarView when a date was selected, as it always displayed and saved the month previous (e.g. if Feb 2nd 2016 was selected, 2/1/2016 would appear in the toast pop-up). To solve this, (month +1) had to be implemented when saving it as a class variable or using it in the toast feature.

Spinner:

The Spinner widget implements a dropdown menu, giving the user a choice of themes for their holiday. When one out of the five themes is clicked, a switch case is used to save a corresponding theme ID as a class variable (e.g. “Sun Holiday” = 0, “Snow Holiday” = 1).

Below is a snippet of code from the onCreate() method in AddHolidayActivity.java. When this file is opened a string array (from strings.xml) of the dropdown items is declared, along with a Spinner widget. Using an ArrayAdapter the Spinner is populated with the items from the string array.

```java
themes = getResources().getStringArray(R.array.theme_list);
spinner = (Spinner)findViewById(R.id.theme_spinner);
ArrayAdapter<String> dataAdapter = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item, themes);
dataAdapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
spinner.setAdapter(dataAdapter);

spinner.setOnItemSelectedListener(new OnItemSelectedListener() {
    @Override
    public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
        switch(position){
            case 0:                 //if the first string is selected
                finalTheme=0;
                break;
            case 1:                 //if the second string is selected
                finalTheme=1;
                break;
            case 2:                 //if the third string is selected
                finalTheme=2;
                break;
            case 3:                 //if the fourth string is selected
                finalTheme=3;
                break;
            case 4:                 //if the fifth string is selected
                finalTheme=4;
                break;
        }
    }
```

Updating

Another widget implemented in *SeeTheCapital* is the SeekBar in the ChangeThemes.java file. Instead of inserting new information, the user has access to the database to update the theme ID of a selected holiday. As the user drags the slider left and right, an ImageView is used to give a preview of all the possible backgrounds (implemented by using a switch case in the onProgressChanged() method). When the slider stops moving, a toast pop-up appears with the name of the current theme (implemented by using a second switch case in the onStopTrackingTouch() method).

When the “save” button is clicked in the ChangeThemes.java activity, the saveTheme() method makes a new instance of the Holiday class, opens the database and calls the updateTheme() method (see below).

```java
public void updateTheme(int newTheme, String holidayName){
    String newThemeStr = newTheme + "";
    ourDatabase.execSQL("UPDATE " + DATABASE_TABLE + " SET " + KEY_THEME + " = " +
            newThemeStr + " WHERE " + KEY_DESTINATION + " LIKE '%"+holidayName+"%';");
}
```

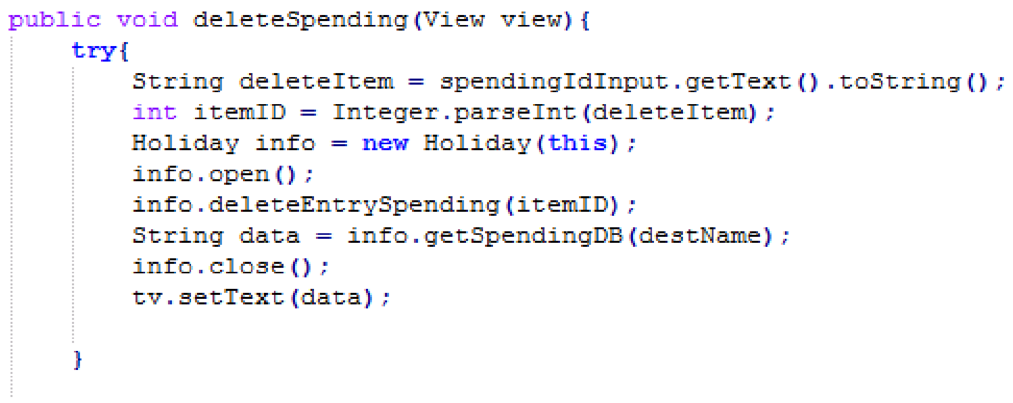
updateTheme() takes in two parameters; the new themeID and the name of the selected holiday (similar to destName in the AddShoppingItem() method above, this variable is also received as extra information with an intent when the activity is opened).

First the method converts the int theme ID into a String, and then it calls an SQL execution statement on the database. The execution is to update the holiday table by setting the old theme ID equal to the new theme ID where the destination name is the same as the holidayName parameter. Pattern matching (Like ‘%holidayName%’) is used in case the user accidentally left a blank space before or after the destination name when inputting the new holiday.

Deleting

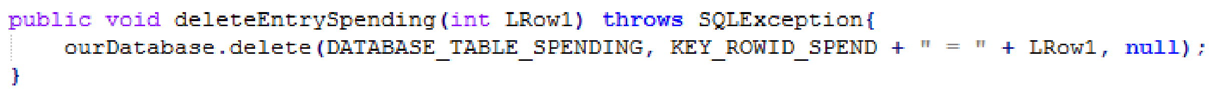
The user is given the option to delete full rows from either the Holiday table, Shopping List table or Spending Money table by simply entering the ID of the item they wish to discard into an EditText box and pressing the “delete” button.

Here is an example of the code used to delete an item from the Spending Money table.



```
public void deleteSpending(View view){
    try{
        String deleteItem = spendingIdInput.getText().toString();
        int itemID = Integer.parseInt(deleteItem);
        Holiday info = new Holiday(this);
        info.open();
        info.deleteEntrySpending(itemID);
        String data = info.getSpendingDB(destName);
        info.close();
        tv.setText(data);

    }
```

deleteSpending() is the method that is called when the “delete” button in HolidaySpendingMoney.java is pressed. The method takes the user input from the EditText box and converts it into an int. A new instance of the Holiday class is made and the database is opened. deleteEntrySpending() is called, which removes the item from the Spending Money table. The method then gets an updated version of the Spending Money table String, sets it in the appropriate TextView, which allows the user to see the new table (with their item deleted).



```
public void deleteEntrySpending(int LRow1) throws SQLException{
    ourDatabase.delete(DATABASE_TABLE_SPENDING, KEY_ROWID_SPEND + " = " + LRow1, null);
}
```

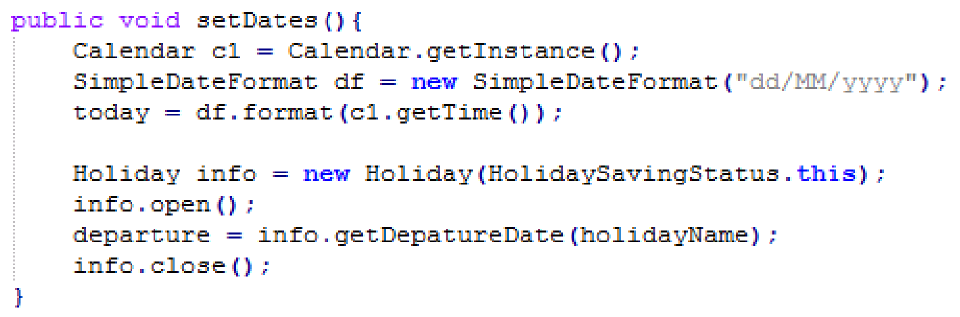
Savings Status

One of the unique features of *SeeTheCapital* as a travel savings app is that it gives the user a breakdown of their savings:

- How much they have in their savings account (using a bank account that runs on SQLite)
- How much the holiday is going to cost them (by adding the total cost of the Shopping List, Spending Money budget and travel price)
- The balance of what the user still needs to save (by subtracting the savings account figure away from the total cost).

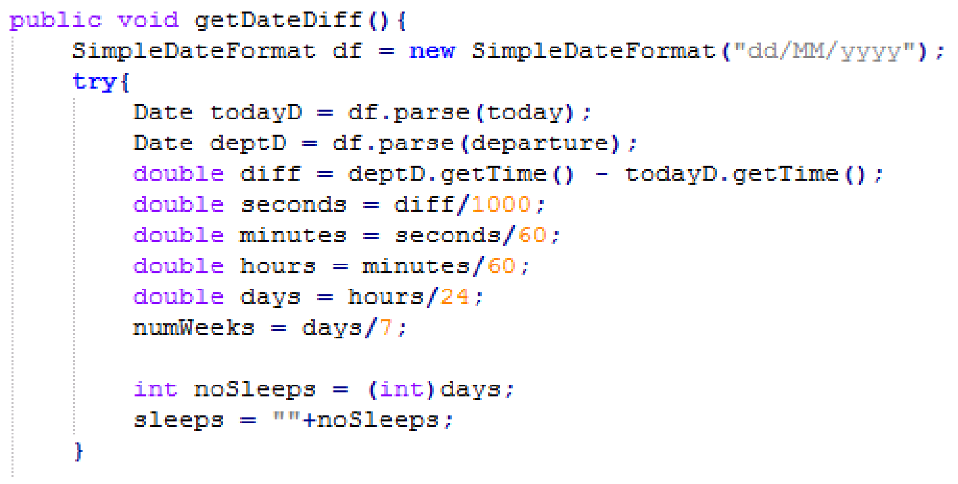
Using basic calculations, SeeTheCapital goes one step further to even tell the user how much they should be saving per week or per day to reach their target.

To do this, the HolidaySavingStatus.java file needs class variables to hold the current date and the departure date of the selected holiday. This is implemented in the setDates() method. The current date is achieved by using a Calendar object and using .getTime(). The departure date is simply extracted from the database.



```java
public void setDates(){
    Calendar c1 = Calendar.getInstance();
    SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");
    today = df.format(c1.getTime());

    Holiday info = new Holiday(HolidaySavingStatus.this);
    info.open();
    departure = info.getDepatureDate(holidayName);
    info.close();
}
```

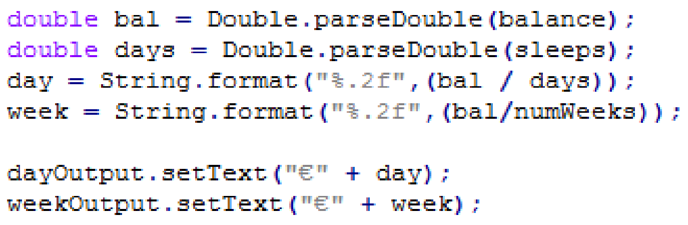
Next, the difference between the two dates is needed (e.g. how many days/weeks until departure). This is implemented in the getDateDiff() method.



```java
public void getDateDiff(){
    SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");
    try{
        Date todayD = df.parse(today);
        Date deptD = df.parse(departure);
        double diff = deptD.getTime() - todayD.getTime();
        double seconds = diff/1000;
        double minutes = seconds/60;
        double hours = minutes/60;
        double days = hours/24;
        numWeeks = days/7;

        int noSleeps = (int)days;
        sleeps = ""+noSleeps;
    }
```

Both the current date and departure date are cast into the data type Date, and the difference in milliseconds is found. That difference is then divided by 1000 to find the difference in seconds; divided by 60 to find the difference in minutes; divided by 60 to find the difference in hours; divided by 24 to find the difference in days; finally divided by 7 to find the difference in weeks.

It is then time to output the result. If the balance of what the user still needed to save is less than or equal to 0 (i.e. they reached their target), then a positive "Congrats!" message is outputted and the per day/per week calculations are no longer needed. Otherwise, the balance is converted into a double and divided by the number of day/weeks, which is then formatted to just two decimal places. The results are saved as class variables and displayed in TextViews to the user.



```
double bal = Double.parseDouble(balance);
double days = Double.parseDouble(sleeps);
day = String.format("%.2f",(bal / days));
week = String.format("%.2f",(bal/numWeeks));

dayOutput.setText("€" + day);
weekOutput.setText("€" + week);
```

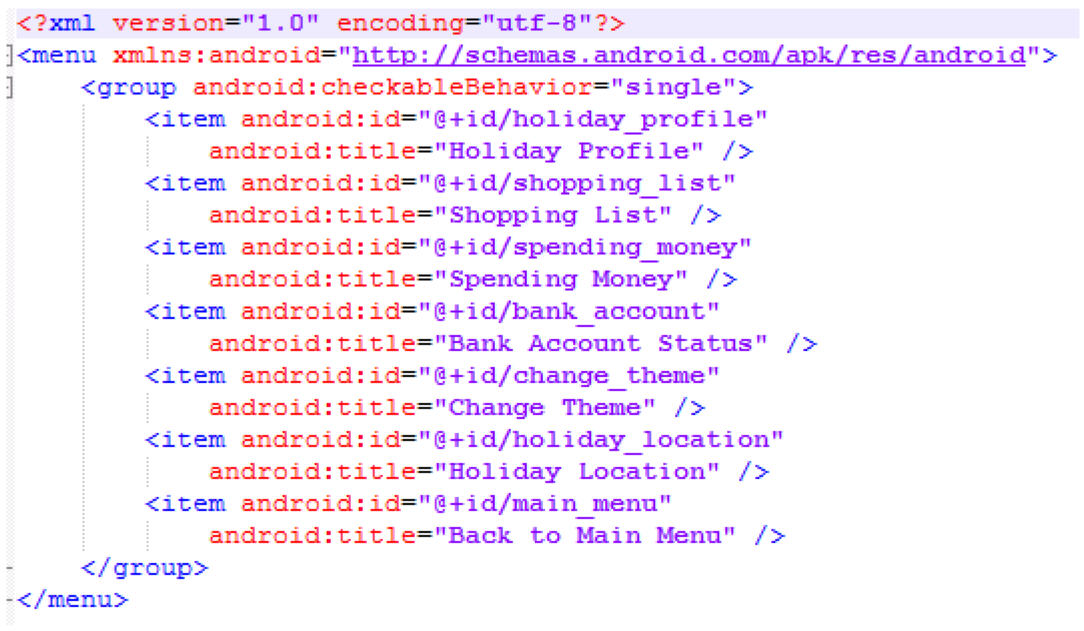
//in onCreate() method

By having a physical countdown to the departure date and a breakdown of the savings target, *SeeTheCapital* helps its customers to put their spending into perspective. If the app suggests the user to save €10 per day before going away, perhaps it will change their decisions when it comes to everyday expenditures (bring a packed lunch into work instead of buying it, spend less on cigarettes/alcohol/clothing per day etc.)

Navigation Drawer

The use of a sliding menu in *SeeTheCapital* is to help the flow of the layout, and to reduce the number of button widgets. The majority of the code is auto-generated by Android Studio when a Navigation Drawer is created.

In the menu folder of the project, an xml file (activity_holiday_profile_page_drawer.xml) is generated and here each item in the menu group corresponds to a clickable option when the Navigation Drawer is open.



```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        <item android:id="@+id/holiday_profile"
            android:title="Holiday Profile" />
        <item android:id="@+id/shopping_list"
            android:title="Shopping List" />
        <item android:id="@+id/spending_money"
            android:title="Spending Money" />
        <item android:id="@+id/bank_account"
            android:title="Bank Account Status" />
        <item android:id="@+id/change_theme"
            android:title="Change Theme" />
        <item android:id="@+id/holiday_location"
            android:title="Holiday Location" />
        <item android:id="@+id/main_menu"
            android:title="Back to Main Menu" />
    </group>
</menu>
```

In the .java file, override methods such as onBackPressed(), onCreateOptionsMenu() [populates the drawer with the above menu group] and onOptionsItemSelected() appear. The code in the onNavigationItemSelected() is manipulated to tell the app what to do when an option is clicked. 'if-else' statements were implemented to bring the user to the corresponding Activity (e.g. if the "Shopping List" option is clicked, the user is brought to the HolidayShoppingList.java activity).

nav_header_holiday_profile_page.xml controls the style of the header. Default TextViews and ImageViews were removed from this file, but the default colour (gradient green) remains.

Each Navigation Drawer activity has their own activity_, app_bar_ and content_ xml files. It is in the content_ xml file that widgets can be added and manipulated.

In total, five Navigation Drawer activities were implemented and together they allow the user to move through the custom holiday profile section of *SeeTheCapital* with ease.

Holiday Location

For the google maps integration we needed to create our layout map fragment. Within the res\layout folder, we created an activity map.xml file and added the following code. We referenced the MapFragment object from the API.

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/frame"
    android:id="@+id/map"

    android:layout_width="match_parent"

    android:layout_height="match_parent"
    android:name="com.google.android.gms.maps.MapFragment"
    tools:context="com.example.sinead.seethecapital.MapsActivity" />
</FrameLayout>
```

Then, within our activity class we set the activity_map.xml as our content view (setContentView(R.layout.map;) and declared a private instance of the Google Map object (private GoogleMap map;). Doing this we could can now manipulate the map object in a similar way as with traditional websites. We set the map object to the MapFragment in the layout file:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_maps);
    // Obtain the SupportMapFragment and get notified when the map is ready to be
used.
    SupportMapFragment mapFragment = (SupportMapFragment)
getSupportFragmentManager()
            .findFragmentById(R.id.map);
    mapFragment.getMapAsync(this);
}
```

With this implemented we could then add markers to the map activity by calling the addMarker method and pass in a MarkerOptions object which includes the latitude and longitude geocode as well as a title, a snippet, and an icon. The snippet is the text that is displayed in the info.Window, which can be attached to an onInfoWindowClick listener. The marker's icon is set to a variable defined in our mipmap resources and the onInfoWindowClick method opens a new activity using an Intent.

```
map.addMarker(new MarkerOptions().position(new LatLng(53.3816, 6.5910))

.title("This is the title")

.snippet("This is the snippet within the InfoWindow")

.icon(BitmapDescriptorFactory

.fromResource(R.mipmap.icon)));
```

```
@Override

public void onInfoWindowClick(Marker marker) {

    Intent intent = new Intent(this, MapsActivity.class);

    intent.putExtra(”snippet”, marker.getSnippet());

    intent.putExtra(”title”, marker.getTitle());

    intent.putExtra(”position”, marker.getPosition());

    startActivity(intent);
```

Finally, Google Maps Android API v2 requires OpenGL ES version 2, a graphics programming interface.

```
<uses-feature
    android:glEsVersion="0x00020000"
    android:required="true"/>
```

The final activity code enables the fully functional Google mapping solution for Android devices.

```
public class MapsActivity extends Activity implements OnInfoWindowClickListener {
    private LatLng defaultLatLng = new LatLng(53.3816, 6.5910);
    private GoogleMap map;
    private int zoomLevel = 7;
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.map);
        try {
            map = ((MapFragment) getFragmentManager().findFragmentById(R.id.map))
                    .getMap();
            if (map!=null){
                map.getUiSettings().setCompassEnabled(true);
                map.setTrafficEnabled(true);
                map.setMyLocationEnabled(true);
```

By adding the map.setTrafficEnabled statement within the java class we can turn on the traffic layer of the GoogleMap V2.

To move the camera instantly to defaultLatLng the below code was needed to be implemented.

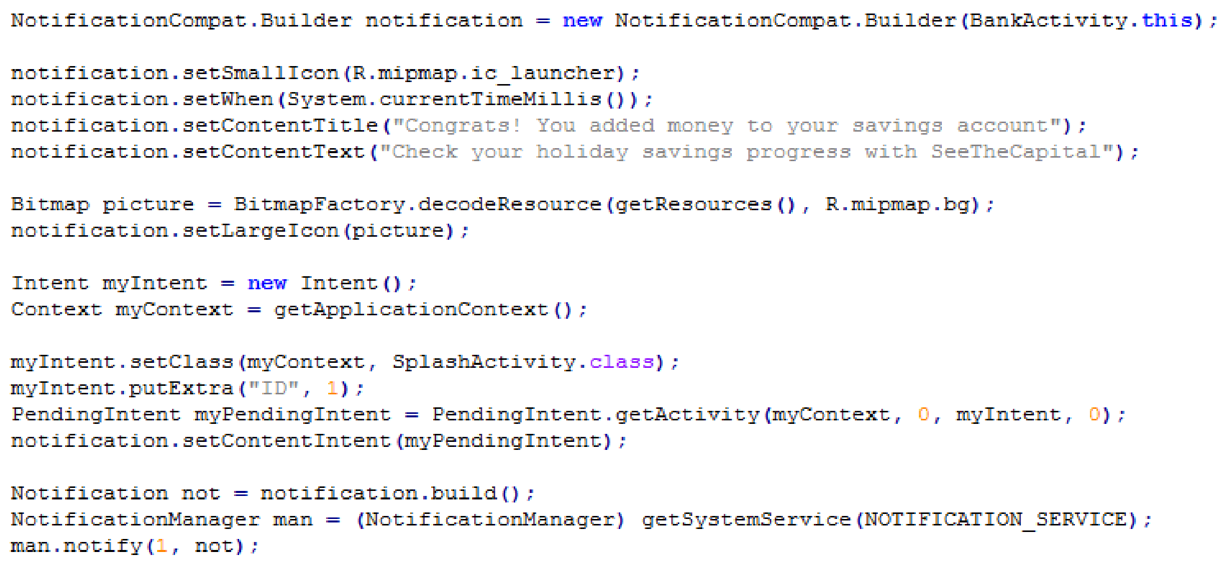
```
map.moveCamera(CameraUpdateFactory.newLatLngZoom(defaultLatLng, zoomLevel));
            map.addMarker(new MarkerOptions().position(defaultLatLng)
                    .title("This is the title")
                    .snippet("This is the snippet within the InfoWindow")
                    .icon(BitmapDescriptorFactory
                            .fromResource(R.drawable.icon)));
            map.setOnInfoWindowClickListener(this);
        }

    }catch (NullPointerException e) {
        e.printStackTrace();
```

Real-World Application

SeeTheCapital uses a simulated bank account using SQLite and basic addition/subtraction SQL update execution statements, yet it has much more potential than that.

The purpose of this android application is to be sold to established banks (such as AIB, Bank of Ireland, Ulster Bank etc.), who have already exposed their own clients to online banking apps. *SeeTheCapital* would be downloadable for free in the app store and would connect directly to a consumer's real-life savings account.

In the current version of the app, an encouraging Notification is sent to the user's mobile device when the "Add" button is pressed in BankActivity.java. It tells the client that they have just increased their savings account and to check on their target progress in the app. Below is the sample of code that sends the Notification to the mobile device, along with the personalised message and icon.



```
NotificationCompat.Builder notification = new NotificationCompat.Builder(BankActivity.this);

notification.setSmallIcon(R.mipmap.ic_launcher);
notification.setWhen(System.currentTimeMillis());
notification.setContentTitle("Congrats! You added money to your savings account");
notification.setContentText("Check your holiday savings progress with SeeTheCapital");

Bitmap picture = BitmapFactory.decodeResource(getResources(), R.mipmap.bg);
notification.setLargeIcon(picture);

Intent myIntent = new Intent();
Context myContext = getApplicationContext();

myIntent.setClass(myContext, SplashActivity.class);
myIntent.putExtra("ID", 1);
PendingIntent myPendingIntent = PendingIntent.getActivity(myContext, 0, myIntent, 0);
notification.setContentIntent(myPendingIntent);

Notification not = notification.build();
NotificationManager man = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
man.notify(1, not);
```

When the notification is clicked by the user, the app opens (from the splash screen) and the notification is destroyed. If this were a feature of a real-life savings app, the client would receive a Notification from the app every time they put money in their savings account (whether it is by standing order, or just a once off). This would remind the user to keep checking their target, their departure date, the items they still need to buy - in other words, to keep using the app in general.

Although *SeeTheCapital* is specifically a travel savings app, it can be a template for many more personal savings trackers (e.g. Christmas savings, House Renovation savings, Back to School savings etc.) and has the potential to be translated into numerous foreign languages, making it an appealing app for anyone, anywhere.

Future Projection

There are many features that were researched into but unfortunately not implemented before the deadline. The following aspects are potential new features for *SeeTheCapital*:

-Save all texts into strings.xml and allow the potential for translation

-Viewing the app when the mobile device is landscape

-A sleeker layout with less overcrowding of widgets in some activities

-Changing more than just the background image with the theme picker (e.g. fonts, colour of Navigation Drawer header)

-An extra activity with the savings breakdown of all holidays put together (in case the user is going on a holiday quickly after their return home from one)

-More encouraging and positive Notifications sent to the mobile device (e.g. when there are only a few sleeps to the user's next holiday, or when the user returns home from a holiday)

-Automatically removing a holiday from the holiday list once the return date has passed

-Tutorial of how to use the app upon the user opening it for the first time