Pieris Kalligeros s1607754

Informatics Large Practical
Coursework 2
Report on the Implementation

Contents

1. Introduction

In the following report I describe the implementation of my design as documented in the Coursework 1 Plan document for the Informatics Practical Course, including any discrepancies between the original design and the implementation.
The topics discussed below include, but are not limited to:

- Algorithms and Data Structures (ADS) used in the implementation & local Persistent Storage and utilization of Cloud FireStore Database
- Design Decisions and discrepancies from plan
- Bonus Features
- Testing of the App
- Acknowledgements and References to sample code used
- Screenshots of the application in use

2. Description of Algorithms and Data Structures (ADS) used for the core functionality of the implementation & Persistent Storage (FireStore/locally)

2.1. Register & Login Activities

2.1.1. Authentication

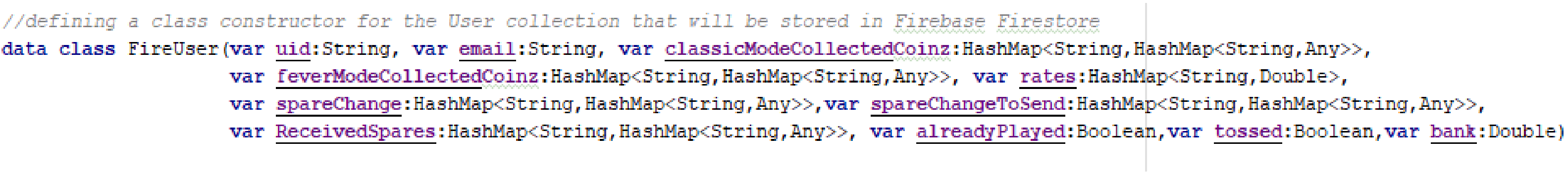
The essential ADS needed to implement the registration and login activities for the application in my implementation were the following:

- Email, Password and Confirm Password text fields and corresponding text variables,
- Register and Sign In buttons with onClickListeners,
- Text Views with onClickListeners to change between the two Activities,
- Firebase Authentication Instance to hold the Authentication data for the current user and communicate with the Firebase Authentication service,
- The Firebase createUserWithEmailAndPassword() method for registration of a new user to the Firebase Authentication service,
- The Firebase signInWithEmailAndPassword() method for signing in of an existing user to the Firebase Authentication service using registered email and password credentials,
- Toast structures to produce error messages to the user upon login or registration.

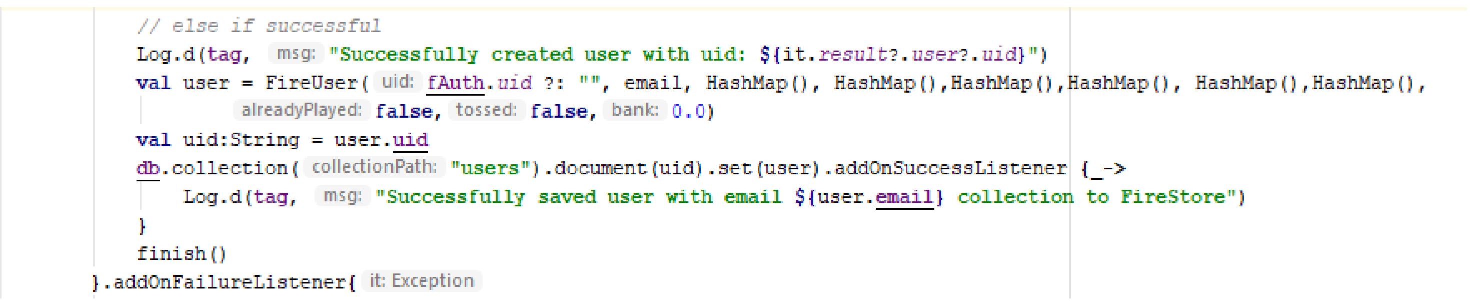
The aforementioned essential structures and methods used for the SignIn and Register Activities can be viewed in the well documented code and are intuitive to understand once the code is read.

2.1.2. FireStore and data class

Apart from these essential structures, I created a Data class named FireUser, in order to hold all the relevant fields with corresponding data types, which were to be stored persistently in the Cloud FireStore service for each user. For each user, there is a collection entry using the auto-generated *user id* (by Firebase Authentication service) as the *key(unique)*, and all fields that require online database storage as the *values*. Note that the fields are stored with their initialized (reset) values e.g. collectedCoinzClassicMode is a null Map initially.
For this separate process that acts as a base for the functionality of the implementation, I enclose 2 code snippets below, taken from the full RegisterActivity code, to show the specific parts of code for this process merged together (rather than having to follow different snippets positioned in different parts of the code:



```
//defining a class constructor for the User collection that will be stored in Firebase Firestore
data class FireUser(var uid:String, var email:String, var classicModeCollectedCoinz:HashMap<String,HashMap<String,Any>>,
                    var feverModeCollectedCoinz:HashMap<String,HashMap<String,Any>>, var rates:HashMap<String,Double>,
                    var spareChange:HashMap<String,HashMap<String,Any>>,var spareChangeToSend:HashMap<String,HashMap<String,Any>>,
                    var ReceivedSpares:HashMap<String,HashMap<String,Any>>, var alreadyPlayed:Boolean,var tossed:Boolean,var bank:Double)
```



```
        // else if successful
        Log.d(tag,  msg: "Successfully created user with uid: ${it.result?.user?.uid}")
        val user = FireUser( uid: fAuth.uid ?: "", email, HashMap(), HashMap(),HashMap(),HashMap(), HashMap(),HashMap(),
                 alreadyPlayed: false,  tossed: false,  bank: 0.0)
        val uid:String = user.uid
        db.collection( collectionPath: "users").document(uid).set(user).addOnSuccessListener {_->
            Log.d(tag,  msg: "Successfully saved user with email ${user.email} collection to FireStore")
        }
        finish()
    }.addOnFailureListener{ it: Exception
```

Figure 1 Initialising FireStore with the default field values

2.2. Main Activity

2.2.1. Authentication

The Main Activity acts as a main menu screen, as well as an activity to download the newest map available from the server using an asynchronous download in the background via Download File Task, as shown in the lectures and to update some persistent storage fields in FireStore, where needed.
The Main Activity first checks if a user is already signed-in (using Firebase Authentication Instance), and if not, it redirects the user to Sign In Activity (using an *intent*) in order to sign in first before being able to play (the user has to be signed in for his personal progress to be stored persistently under his user collection in FireStore).

2.2.2. JSON parsing and FireStore storage interaction

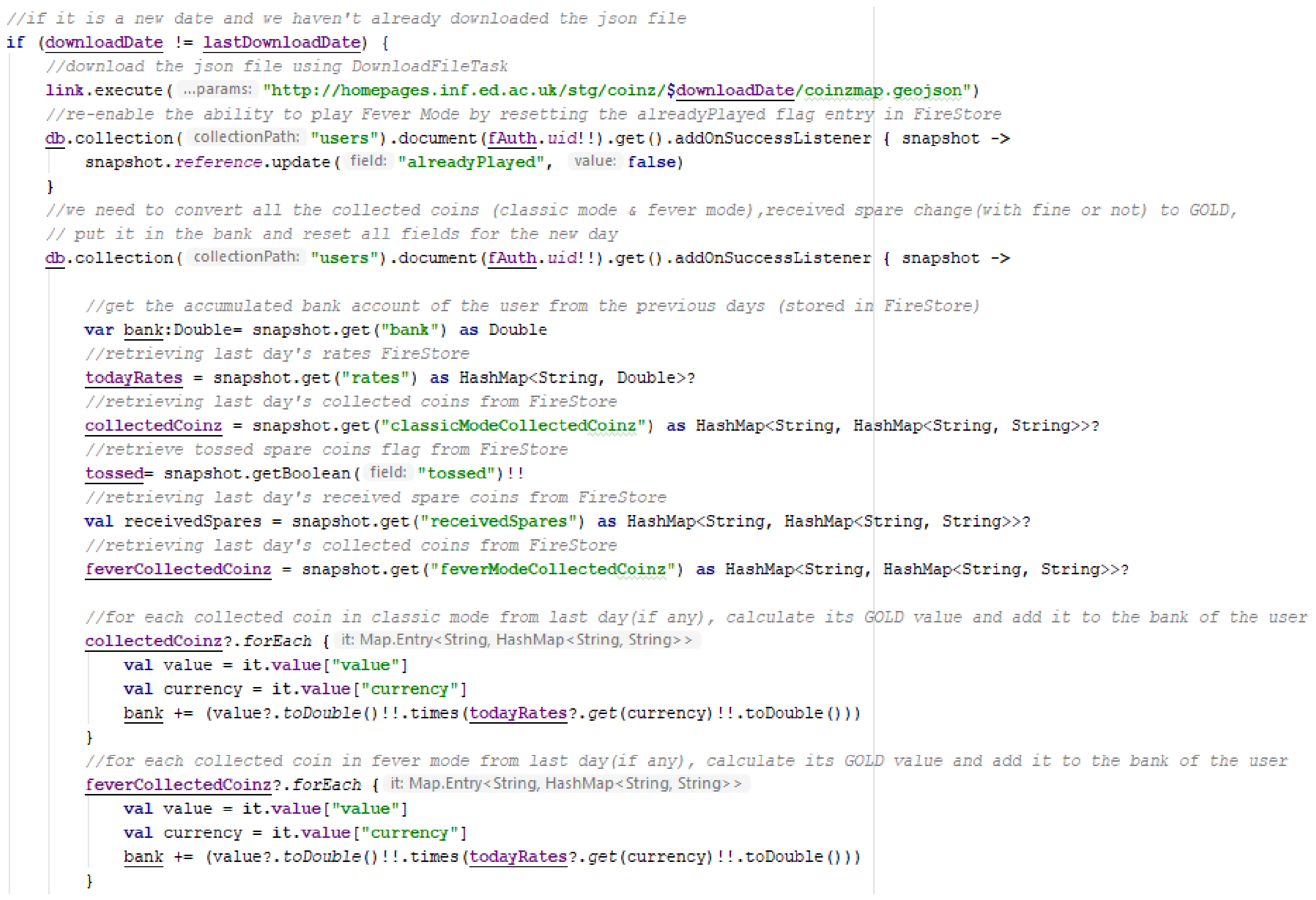
The GEOJSON map is *not fully* parsed within the Main Activity, as the display and collection of coins in the game happens in the two play modes: Classic Mode and Coin Fever Mode (see *4. Bonus Features*)
The only features that are *parsed* from the GeoJSON file in Main Activity are the *rates* of the 4 currencies for today's map, in order to update them prior to gameplay (this happens *after conversion of previous day's coins, read below)*.

Furthermore, the Main Activity interacts with the FireStore service in order to retrieve the previous day's *rates, collected coins for the two modes, received spare change and bank balance* (if any), in order to update the bank balance with the total new GOLD value of the converted coins of the previous day using their corresponding *rates* (if any) for the current user.
The currency rates are stored using a variable <u>todayRates</u> of type **HashMap<String,Double>,** with each key being the currency <u>name</u> and each value being the current day's <u>rate</u> of that currency(double for long precision).
This process takes place when a new day arrives (lastDownloadDate != downloadDate), following the plan specification of Coursework 1's *design decisions,* in order to automatically convert all collected coins and received spare change to GOLD when a new day arrives, rather than the user having to *bank* collected coins and received spare change explicitly, one-by-one. After the conversion and update of the bank is finished, it *resets* all fields in FireStore and updates the *rates* field with today's currency rates.

Moreover, the Main Activity requests for the *alreadyPlayed (see section 3.2.)* and *tossed (see section 4.3.)* **boolean** flag fields from FireStore, and if they are *true*, it acts accordingly (restricts the user from playing Coin Fever Mode and applies a 75% fine on received spare coins' conversion, respectively), or *resets* the flag values if a new day has arrived.

In order to make the above description clearer, a snippet from the Main Activity code is shown in the next page with the interaction of Main Activity with the FireStore service and the appropriate updating of variables and fields.

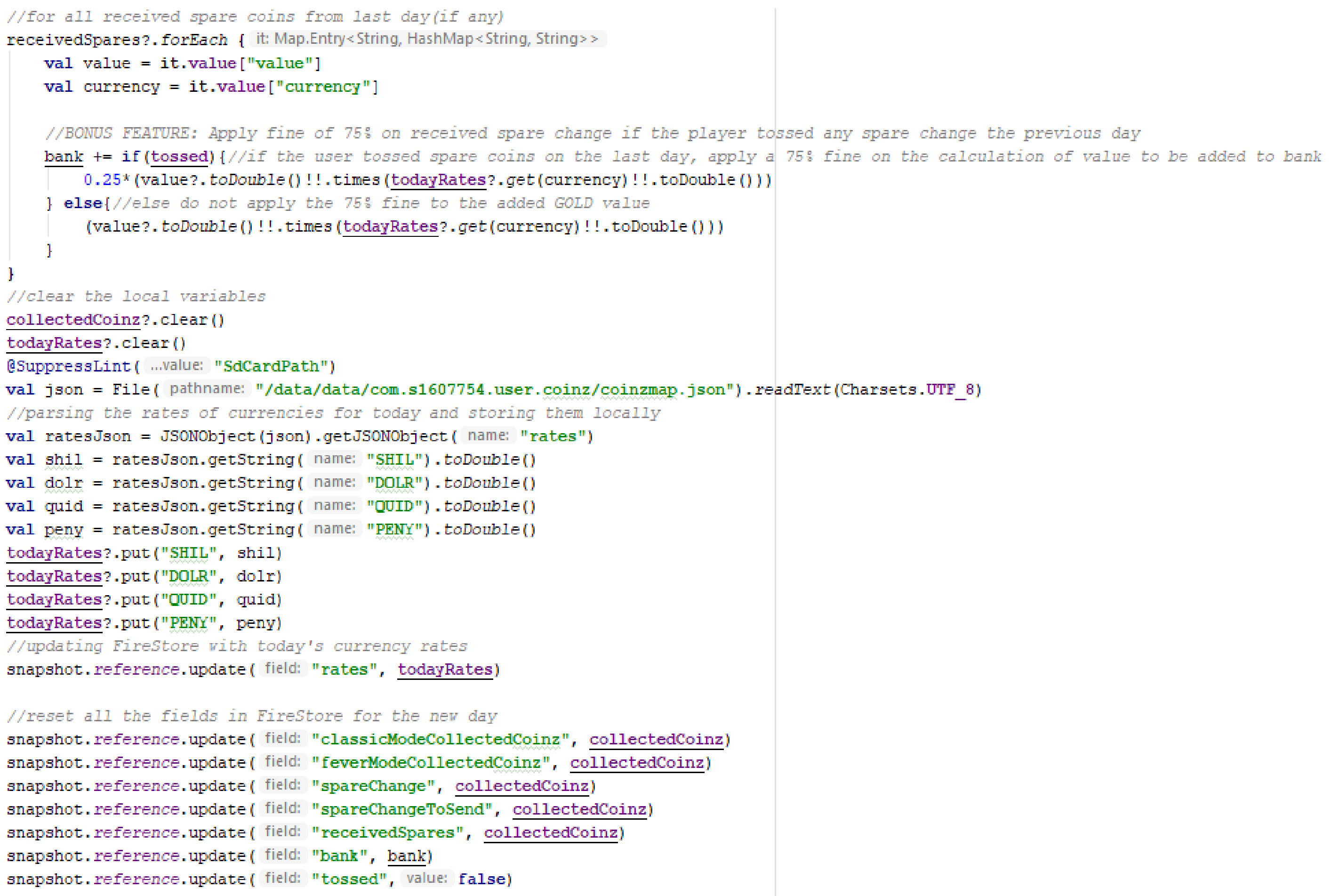
The rest of Main Activity is mostly comprised of onClickListeners on buttons from the view layout, each leading to a corresponding Activity using intents, along with a Sign Out button which first uses Firebase Authentication's *signout()* method to sign the user out before redirecting them to Sign In Activity.



```
//if it is a new date and we haven't already downloaded the json file
if (downloadDate != lastDownloadDate) {
    //download the json file using DownloadFileTask
    link.execute( ...params: "http://homepages.inf.ed.ac.uk/stg/coinz/$downloadDate/coinzmap.geojson")
    //re-enable the ability to play Fever Mode by resetting the alreadyPlayed flag entry in FireStore
    db.collection( collectionPath: "users").document(fAuth.uid!!).get().addOnSuccessListener { snapshot ->
        snapshot.reference.update( field: "alreadyPlayed",  value: false)
    }
    //we need to convert all the collected coins (classic mode & fever mode),received spare change(with fine or not) to GOLD,
    // put it in the bank and reset all fields for the new day
    db.collection( collectionPath: "users").document(fAuth.uid!!).get().addOnSuccessListener { snapshot ->

        //get the accumulated bank account of the user from the previous days (stored in FireStore)
        var bank:Double= snapshot.get("bank") as Double
        //retrieving last day's rates FireStore
        todayRates = snapshot.get("rates") as HashMap<String, Double>?
        //retrieving last day's collected coins from FireStore
        collectedCoinz = snapshot.get("classicModeCollectedCoinz") as HashMap<String, HashMap<String, String>>?
        //retrieve tossed spare coins flag from FireStore
        tossed= snapshot.getBoolean( field: "tossed")!!
        //retrieving last day's received spare coins from FireStore
        val receivedSpares = snapshot.get("receivedSpares") as HashMap<String, HashMap<String, String>>?
        //retrieving last day's collected coins from FireStore
        feverCollectedCoinz = snapshot.get("feverModeCollectedCoinz") as HashMap<String, HashMap<String, String>>?

        //for each collected coin in classic mode from last day(if any), calculate its GOLD value and add it to the bank of the user
        collectedCoinz?.forEach { it: Map.Entry<String, HashMap<String, String>>
            val value = it.value["value"]
            val currency = it.value["currency"]
            bank += (value?.toDouble()!!.times(todayRates?.get(currency)!!.toDouble()))
        }
        //for each collected coin in fever mode from last day(if any), calculate its GOLD value and add it to the bank of the user
        feverCollectedCoinz?.forEach { it: Map.Entry<String, HashMap<String, String>>
            val value = it.value["value"]
            val currency = it.value["currency"]
            bank += (value?.toDouble()!!.times(todayRates?.get(currency)!!.toDouble()))
        }
```



```
        //for all received spare coins from last day(if any)
        receivedSpares?.forEach { it: Map.Entry<String, HashMap<String, String>>
            val value = it.value["value"]
            val currency = it.value["currency"]

            //BONUS FEATURE: Apply fine of 75% on received spare change if the player tossed any spare change the previous day
            bank += if(tossed){//if the user tossed spare coins on the last day, apply a 75% fine on the calculation of value to be added to bank
                0.25*(value?.toDouble()!!.times(todayRates?.get(currency)!!.toDouble()))
            } else{//else do not apply the 75% fine to the added GOLD value
                (value?.toDouble()!!.times(todayRates?.get(currency)!!.toDouble()))
            }
        }
        //clear the local variables
        collectedCoinz?.clear()
        todayRates?.clear()
        @SuppressLint( ...value: "SdCardPath")
        val json = File( pathname: "/data/data/com.s1607754.user.coinz/coinzmap.json").readText(Charsets.UTF_8)
        //parsing the rates of currencies for today and storing them locally
        val ratesJson = JSONObject(json).getJSONObject( name: "rates")
        val shil = ratesJson.getString( name: "SHIL").toDouble()
        val dolr = ratesJson.getString( name: "DOLR").toDouble()
        val quid = ratesJson.getString( name: "QUID").toDouble()
        val peny = ratesJson.getString( name: "PENY").toDouble()
        todayRates?.put("SHIL", shil)
        todayRates?.put("DOLR", dolr)
        todayRates?.put("QUID", quid)
        todayRates?.put("PENY", peny)
        //updating FireStore with today's currency rates
        snapshot.reference.update( field: "rates", todayRates)

        //reset all the fields in FireStore for the new day
        snapshot.reference.update( field: "classicModeCollectedCoinz", collectedCoinz)
        snapshot.reference.update( field: "feverModeCollectedCoinz", collectedCoinz)
        snapshot.reference.update( field: "spareChange", collectedCoinz)
        snapshot.reference.update( field: "spareChangeToSend", collectedCoinz)
        snapshot.reference.update( field: "receivedSpares", collectedCoinz)
        snapshot.reference.update( field: "bank", bank)
        snapshot.reference.update( field: "tossed", value: false)
```

2.3. Classic Mode Activity

2.3.1 General Description

The Classic Mode Activity is responsible for the core functionality of the application and the main gameplay specified in the coursework specification (coin collection, bank account, sending spare change).

The Classic Mode Activity (and Coin Fever Mode Activity, see 2.4.) also *updates the last download date* to today's date as shown in the lecture slides, using the *My Preferences* file. This is not done in the Main Activity because the user has to actually play a session of a game mode of Coinz for the day data to change to today's date (apart from resetting the data which is done in Main Activity).

It functions by displaying a MapBox map view initialised to the Central Campus Area, parsing the coins from the GeoJSON file, matching their currency with the corresponding color icon, displaying them on the map and allowing the user to traverse the map by using his device's GPS location updates to move along the map area, centering the camera to his current position.

The MapBox functionalities, displays and lifecycle methods are implemented as shown in the lectures, using a LocationEngineListener, a locationLayerPlugin and an OnMapReadyCallback.
Note: Some of the above MapBox methods are now *deprecated*. I attempted to use the updated MapBox *location component* object, however it did not include a LocationEngineListener and an onLocationChanged Callback, which I needed for my implementation of *automatic coin collection,* and hence I remained with the outdated lecture slides version of the MapBoxSDK 6.5.0. objects.

2.3.2. Parsing of coins from the locally stored GeoJSON file

Coins are parsed from the *locally stored GeoJSON file (download was completed in Main Activity)* is done within my *loadMarkers()* method, which is invoked inside the onMapReady Callback method *prior* to updating the map with the parsed *markers (markeropts)*. The process of parsing the features from the JSON file and breaking down each coin's properties is done as explained in the lecture slides *"maps and navigation"* (feature collection *fc*-feature *f*, position *p*, geometry *g* etc.).

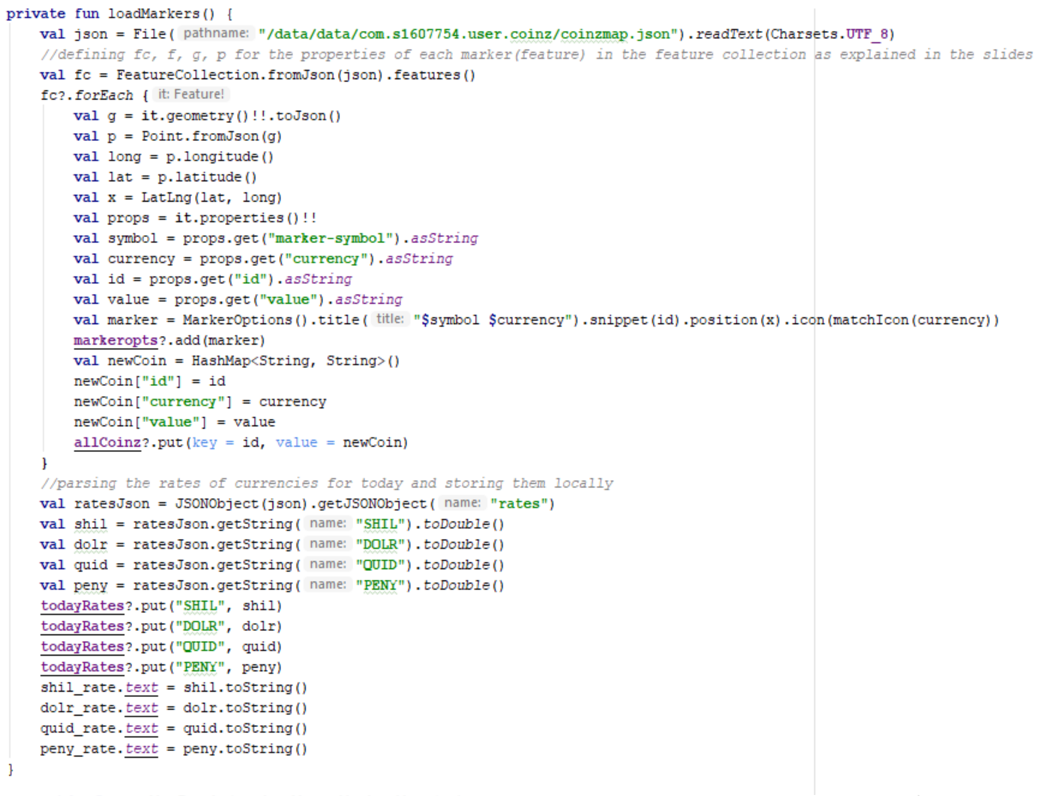
Each coin's currency is matched with the correct corresponding icon color using my *matchIcon()* method.

Each coin is represented as a **HashMap<String,Any>** with its *id* as the key and its *id,value,currency and position* as the values(used type Any to accommodate multiple data types).

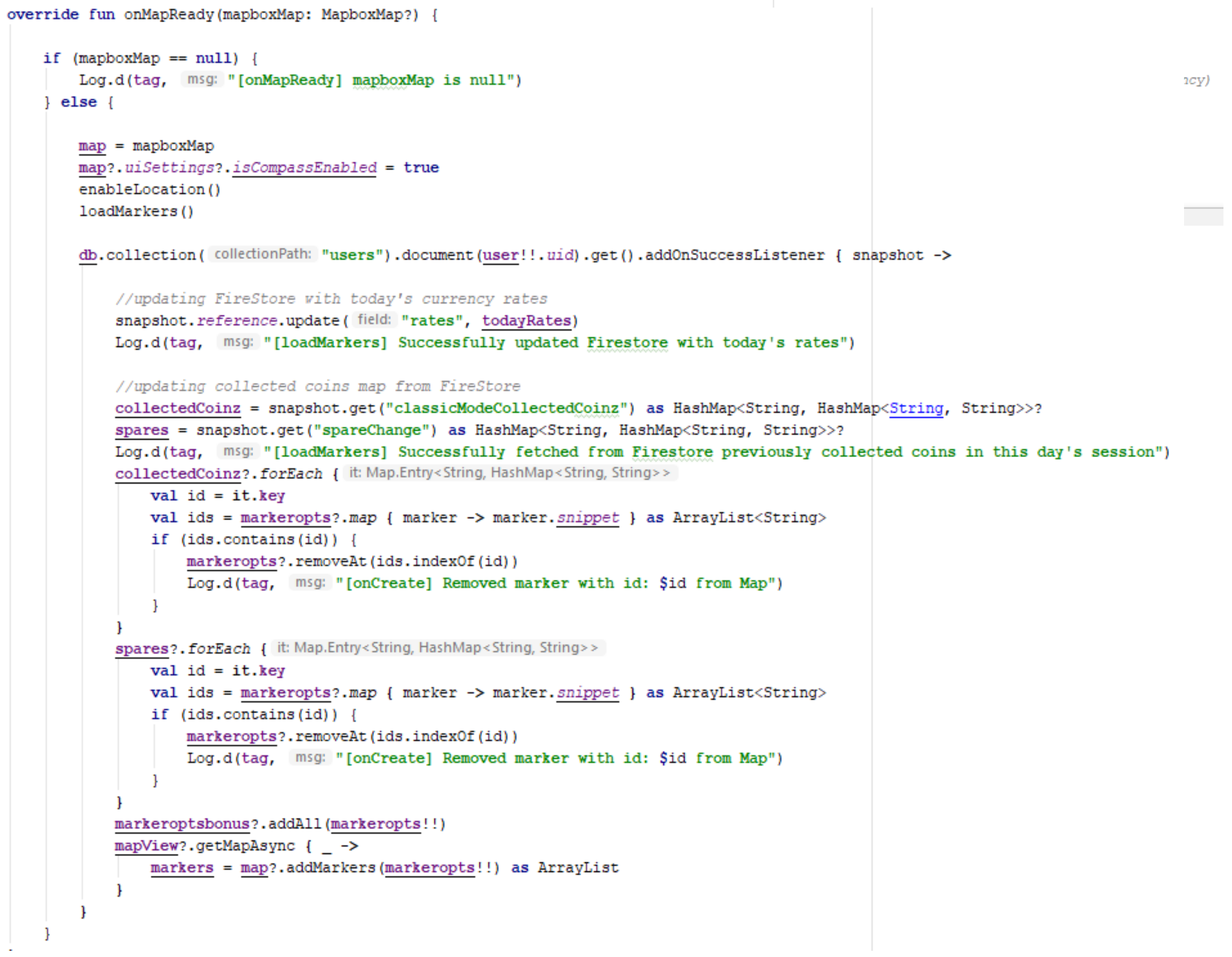
Each coin *marker* is represented as a **MarkerOptions** object, containing properties such as a position, title, snippet and *icon.* All markers are stored in an **ArrayList<MarkerOptions>** variable called *markeropts,* and loaded in the map using *addMarkers()* method.

The *currency rates* for all coins today are also *re-parsed* (couldn't send all currency rates from Main Activity) and displayed as a *legend* on top of the map view of the Activity. After the *loadMarkers()* method is done (still within onMapReady), a request is sent to FireStore to retrieve any data on *collected coins from any previous sessions within the same day (collectedCoinz* and *spareChange).* Such coins are removed from the marker list (*markeropts)* before **rendering the map** using an AsyncTask.

Snippets from the code for parsing of the GeoJSON file and rendering the map is shown in the next page:



```
private fun loadMarkers() {
    val json = File( pathname: "/data/data/com.s1607754.user.coinz/coinzmap.json").readText(Charsets.UTF_8)
    //defining fc, f, g, p for the properties of each marker(feature) in the feature collection as explained in the slides
    val fc = FeatureCollection.fromJson(json).features()
    fc?.forEach { it: Feature!
        val g = it.geometry()!!.toJson()
        val p = Point.fromJson(g)
        val long = p.longitude()
        val lat = p.latitude()
        val x = LatLng(lat, long)
        val props = it.properties()!!
        val symbol = props.get("marker-symbol").asString
        val currency = props.get("currency").asString
        val id = props.get("id").asString
        val value = props.get("value").asString
        val marker = MarkerOptions().title( title: "$symbol $currency").snippet(id).position(x).icon(matchIcon(currency))
        markeropts?.add(marker)
        val newCoin = HashMap<String, String>()
        newCoin["id"] = id
        newCoin["currency"] = currency
        newCoin["value"] = value
        allCoinz?.put(key = id, value = newCoin)
    }
    //parsing the rates of currencies for today and storing them locally
    val ratesJson = JSONObject(json).getJSONObject( name: "rates")
    val shil = ratesJson.getString( name: "SHIL").toDouble()
    val dolr = ratesJson.getString( name: "DOLR").toDouble()
    val quid = ratesJson.getString( name: "QUID").toDouble()
    val peny = ratesJson.getString( name: "PENY").toDouble()
    todayRates?.put("SHIL", shil)
    todayRates?.put("DOLR", dolr)
    todayRates?.put("QUID", quid)
    todayRates?.put("PENY", peny)
    shil_rate.text = shil.toString()
    dolr_rate.text = dolr.toString()
    quid_rate.text = quid.toString()
    peny_rate.text = peny.toString()
}
```



```
override fun onMapReady(mapboxMap: MapboxMap?) {

    if (mapboxMap == null) {
        Log.d(tag,  msg: "[onMapReady] mapboxMap is null")
    } else {

        map = mapboxMap
        map?.uiSettings?.isCompassEnabled = true
        enableLocation()
        loadMarkers()

        db.collection( collectionPath: "users").document(user!!.uid).get().addOnSuccessListener { snapshot ->

            //updating FireStore with today's currency rates
            snapshot.reference.update( field: "rates", todayRates)
            Log.d(tag,  msg: "[loadMarkers] Successfully updated Firestore with today's rates")

            //updating collected coins map from FireStore
            collectedCoinz = snapshot.get("classicModeCollectedCoinz") as HashMap<String, HashMap<String, String>>?
            spares = snapshot.get("spareChange") as HashMap<String, HashMap<String, String>>?
            Log.d(tag,  msg: "[loadMarkers] Successfully fetched from Firestore previously collected coins in this day's session")
            collectedCoinz?.forEach { it: Map.Entry<String, HashMap<String, String>>
                val id = it.key
                val ids = markeropts?.map { marker -> marker.snippet } as ArrayList<String>
                if (ids.contains(id)) {
                    markeropts?.removeAt(ids.indexOf(id))
                    Log.d(tag,  msg: "[onCreate] Removed marker with id: $id from Map")
                }
            }
            spares?.forEach { it: Map.Entry<String, HashMap<String, String>>
                val id = it.key
                val ids = markeropts?.map { marker -> marker.snippet } as ArrayList<String>
                if (ids.contains(id)) {
                    markeropts?.removeAt(ids.indexOf(id))
                    Log.d(tag,  msg: "[onCreate] Removed marker with id: $id from Map")
                }
            }
            markeroptsbonus?.addAll(markeropts!!)
            mapView?.getMapAsync { _ ->
                markers = map?.addMarkers(markeropts!!) as ArrayList
            }
        }
    }
```

2.3.3. Coin Collection

The collection of coins happens *automatically* when the user reaches a radius of <=25 metres away from a coin's *position* (LatLng). All coins on the map (*markers list)* are automatically checked for their distance to the *updated current user's location* within the *onLocationChanged()* callback method, with collection also taking place within that method.

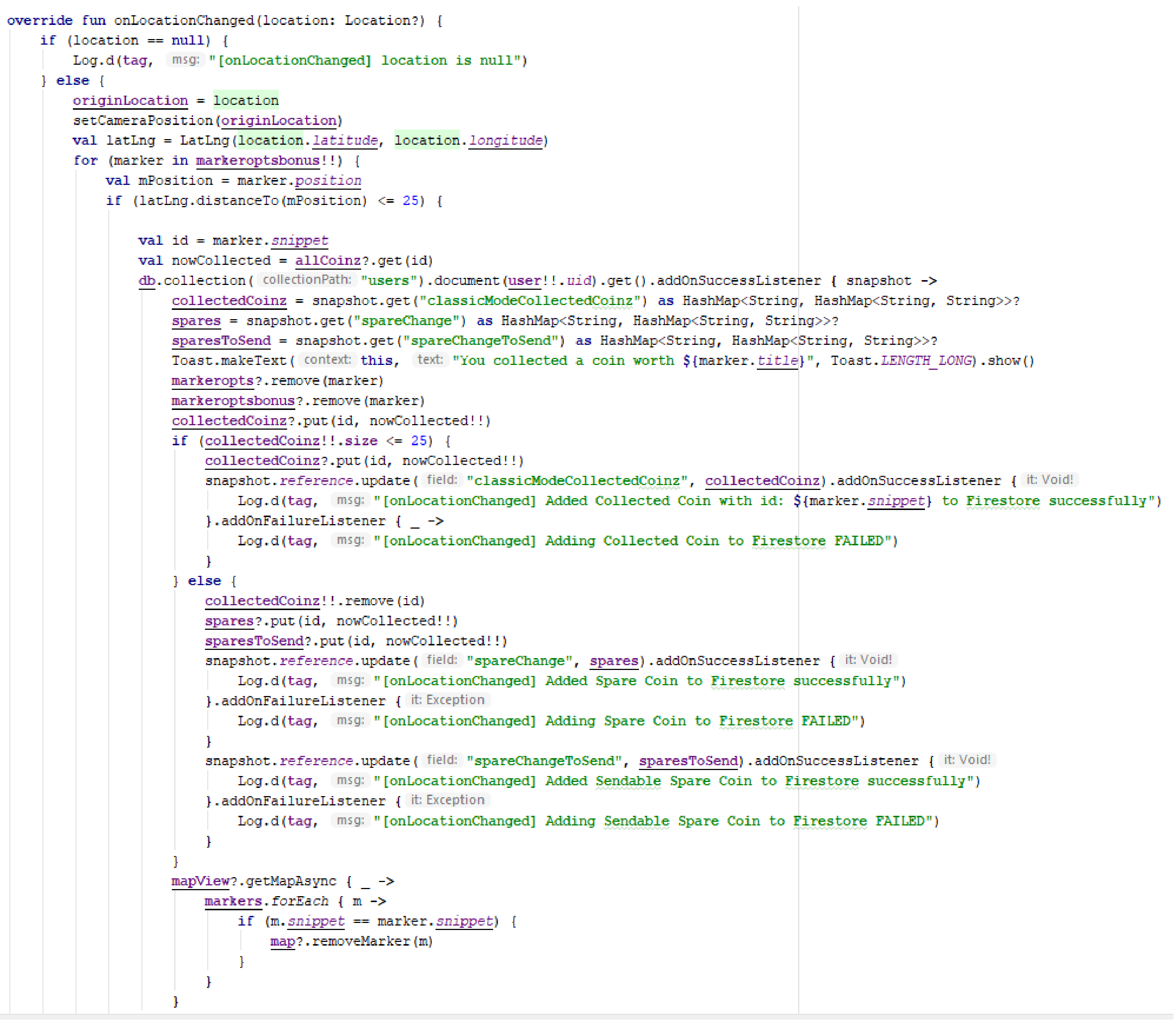
A message (toast) is displayed to the user when he collects a coin, indicating the *integer rounded value* and *currency* of the coin (see 7. Screenshots).

As with parsing, a *collected coin* is represented as a **HashMap<String,Any>,** while the total *collected coins* and *spare change(separate)* are represented as **HashMap<String, HashMap<String,Any>>** variables**,** where the keys are the ids of the coins and the values are the *coin* representations.
This peculiar representation was used as a result of some FireStore restrictions on the data structures that it can store on the Cloud database (no nested *Arrays* or *Lists*), so the only possible solution I thought to use was *nested Maps (used HashMaps* to avoid <u>duplicates)</u>.

Upon collecting a coin, the method retrieves the latest *collectedCoinz* and *spareChange* fields from the user's data on FireStore and updates *one of the two* fields by adding the new collected coin. It conditionally adds the collected coin to the *collectedCoinz* field if the field's size is <25 (i.e. the user has collected *less than* 25 coins today), or in the *spareChange* field otherwise (i.e. the user has collected 25 *or more* out of 50 coins today). The relevant field is then updated and persistently stored on FireStore with the newly added coins. This ensures that in the next *same day session* of Classic Mode, the user will *not* be able to re-collect the same coins twice, as the collected coins and spare change will be retrieved from FireStore and *removed from the map prior to rendering it* and displaying it to the user.

Whilst collecting a *spare* coin, that coin is also added in another similar **HashMap** collection, *spareChangeToSend*, in order to track the sent/unsent spare coins *without* allowing coins to be displayed *after collection by the user.* This variable is used in the next activity, SpareActivity for the user to *send spare change to another user*.

Snippets from the *onLocationChanged()* method, which includes the collection of coins, are displayed in the next page (the rest of the code can be viewed in the application project in Android studio):



```
override fun onLocationChanged(location: Location?) {
    if (location == null) {
        Log.d(tag,  msg: "[onLocationChanged] location is null")
    } else {
        originLocation = location
        setCameraPosition(originLocation)
        val latLng = LatLng(location.latitude, location.longitude)
        for (marker in markeroptsbonus!!) {
            val mPosition = marker.position
            if (latLng.distanceTo(mPosition) <= 25) {

                val id = marker.snippet
                val nowCollected = allCoinz?.get(id)
                db.collection( collectionPath: "users").document(user!!.uid).get().addOnSuccessListener { snapshot ->
                    collectedCoinz = snapshot.get("classicModeCollectedCoinz") as HashMap<String, HashMap<String, String>>?
                    spares = snapshot.get("spareChange") as HashMap<String, HashMap<String, String>>?
                    sparesToSend = snapshot.get("spareChangeToSend") as HashMap<String, HashMap<String, String>>?
                    Toast.makeText( context: this,  text: "You collected a coin worth ${marker.title}", Toast.LENGTH_LONG).show()
                    markeropts?.remove(marker)
                    markeroptsbonus?.remove(marker)
                    collectedCoinz?.put(id, nowCollected!!)
                    if (collectedCoinz!!.size <= 25) {
                        collectedCoinz?.put(id, nowCollected!!)
                        snapshot.reference.update( field: "classicModeCollectedCoinz", collectedCoinz).addOnSuccessListener { it: Void!
                            Log.d(tag,  msg: "[onLocationChanged] Added Collected Coin with id: ${marker.snippet} to Firestore successfully")
                        }.addOnFailureListener { _ ->
                            Log.d(tag,  msg: "[onLocationChanged] Adding Collected Coin to Firestore FAILED")
                        }
                    } else {
                        collectedCoinz!!.remove(id)
                        spares?.put(id, nowCollected!!)
                        sparesToSend?.put(id, nowCollected!!)
                        snapshot.reference.update( field: "spareChange", spares).addOnSuccessListener { it: Void!
                            Log.d(tag,  msg: "[onLocationChanged] Added Spare Coin to Firestore successfully")
                        }.addOnFailureListener { it: Exception
                            Log.d(tag,  msg: "[onLocationChanged] Adding Spare Coin to Firestore FAILED")
                        }
                        snapshot.reference.update( field: "spareChangeToSend", sparesToSend).addOnSuccessListener { it: Void!
                            Log.d(tag,  msg: "[onLocationChanged] Added Sendable Spare Coin to Firestore successfully")
                        }.addOnFailureListener { it: Exception
                            Log.d(tag,  msg: "[onLocationChanged] Adding Sendable Spare Coin to Firestore FAILED")
                        }
                    }
                    mapView?.getMapAsync { _ ->
                        markers.forEach { m ->
                            if (m.snippet == marker.snippet) {
                                map?.removeMarker(m)
                            }
                        }
                    }
```

2.4. Spare Activity

2.4.1. General Information

The Spare Activity is responsible for sending *spare change coins* to *another user* (error checks included for restricting sending coins to self or to a non-existing user).

The activity is only accessible through the App Bar (see *3.2. Extra Implementations*) in *Classic Mode Activity*, since the feature of sending spare change is only available in that mode of the game.

The user also has the option of *tossing his spare change*, which, however, causes a 75% fine to be imposed on any spare coins he/she might *receive* from another user for that day. (see *4. Bonus Features*).

2.4.2. ADS used and FireStore interaction

The Spare Activity allows the user to input a user e-mail and press a button (onClickListener) to send his/her spare change to him/her. The Activity also displays the amount of spare change that the user is able to spend and produces Toast messages (see 7. Screenshots) if the user tries to send spare change while not having *any spare change available to send*.

The spare change to send is represented using a **HashMap<String,HashMap<String,Any>>**, similar to the Classic Mode Data Structures used. This data is stored persistently in the FireStore database.
Each user's collection also has a field *receivedSpares* which is of similar data type and holds any *received spare change* for the current day.

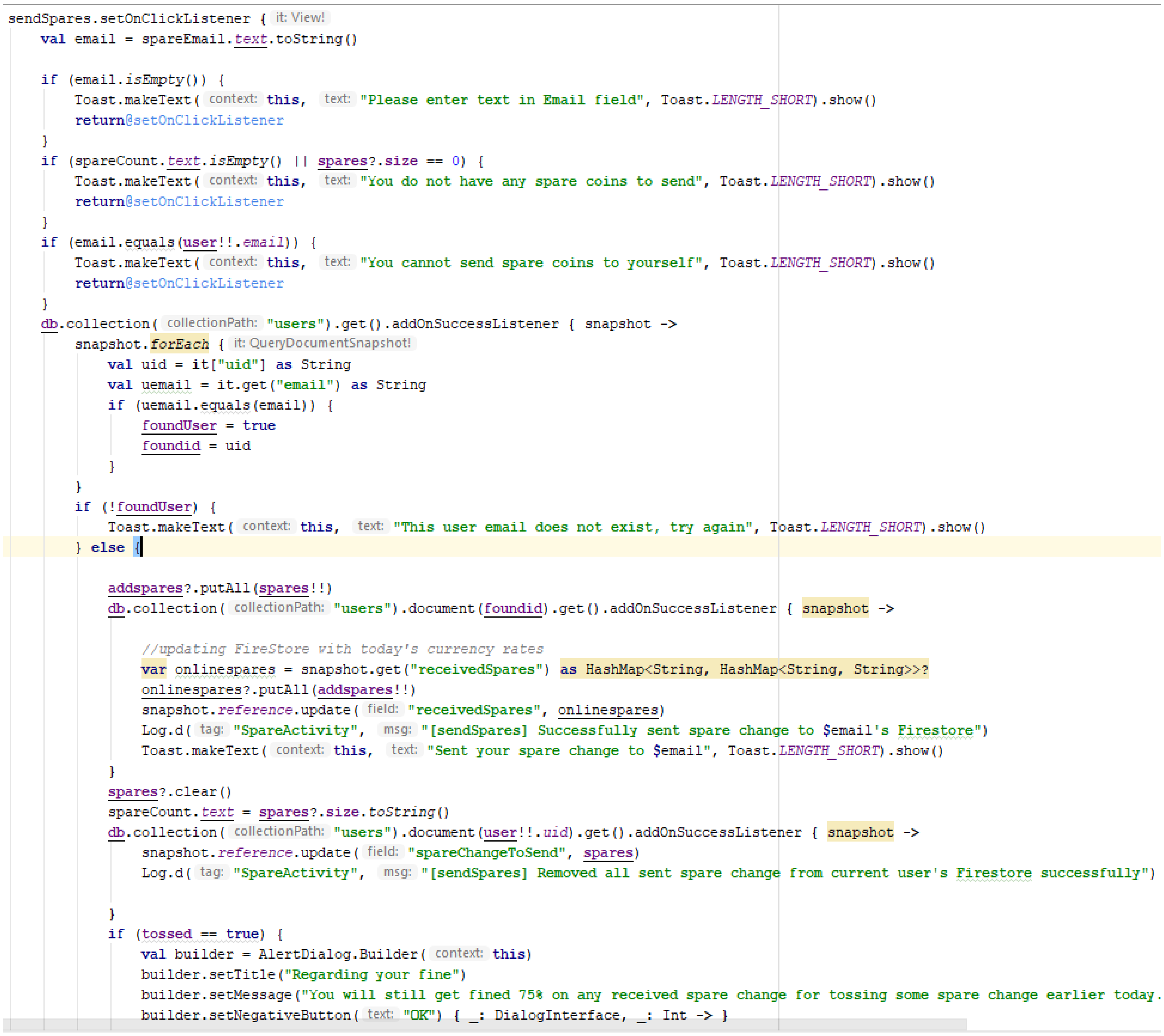
Note: *As per the design decisions in Coursework 1, any spare coins received can also be the same with other collected coins and it will still be banked into the user's account as GOLD.*

The Activity then checks (assuming the user has some spare change to send) for the e-mail address that the user input among the FireStore *users* collection, and if it finds the corresponding user, it *adds* the spare change from the *current user* to the *receivedSpares* field of the target user.
The activity finally removes those spare coins from the *spareChangeToSend* field of the current user, so that he/she cannot send the same spare coin multiple times.

If the activity *doesn't find* the target user, it produces a toast message to the user that the entered e-mail was not found (see 7. Screenshots).
Another error check that the Activity performs is if the input e-mail is the current user's e-mail. The Activity restricts the user from sending spare change to himself/herself and produces a relevant toast message in that event (see 7. Screenshots).

The user can also press the *Toss Spare Change button* if he/she wishes to toss the collected spare change, meaning throw them away without sending it to another user. The activity produces an *Alert dialog*, warning the user that if he/she *tosses* his spare change, a *fine will be imposed* on any *received spare change* value for that specific day (see *4.3 Fair Play Fine*). Of course, if the user hasn't got *any* spare change the activity only produces a toast message to the user informing them so.

A Snippet for *sending* spare change from the Spare Activity code is shown below: (the rest of the code, including *tossing* spare change, can be reviewed in the Android Studio project files).



```
sendSpares.setOnClickListener { it: View!
    val email = spareEmail.text.toString()

    if (email.isEmpty()) {
        Toast.makeText( context: this,  text: "Please enter text in Email field", Toast.LENGTH_SHORT).show()
        return@setOnClickListener
    }
    if (spareCount.text.isEmpty() || spares?.size == 0) {
        Toast.makeText( context: this,  text: "You do not have any spare coins to send", Toast.LENGTH_SHORT).show()
        return@setOnClickListener
    }
    if (email.equals(user!!.email)) {
        Toast.makeText( context: this,  text: "You cannot send spare coins to yourself", Toast.LENGTH_SHORT).show()
        return@setOnClickListener
    }
    db.collection( collectionPath: "users").get().addOnSuccessListener { snapshot ->
        snapshot.forEach { it: QueryDocumentSnapshot!
            val uid = it["uid"] as String
            val uemail = it.get("email") as String
            if (uemail.equals(email)) {
                foundUser = true
                foundid = uid
            }
        }
        if (!foundUser) {
            Toast.makeText( context: this,  text: "This user email does not exist, try again", Toast.LENGTH_SHORT).show()
        } else {

            addspares?.putAll(spares!!)
            db.collection( collectionPath: "users").document(foundid).get().addOnSuccessListener { snapshot ->

                //updating FireStore with today's currency rates
                var onlinespares = snapshot.get("receivedSpares") as HashMap<String, HashMap<String, String>>?
                onlinespares?.putAll(addspares!!)
                snapshot.reference.update( field: "receivedSpares", onlinespares)
                Log.d( tag: "SpareActivity",  msg: "[sendSpares] Successfully sent spare change to $email's Firestore")
                Toast.makeText( context: this,  text: "Sent your spare change to $email", Toast.LENGTH_SHORT).show()
            }
            spares?.clear()
            spareCount.text = spares?.size.toString()
            db.collection( collectionPath: "users").document(user!!.uid).get().addOnSuccessListener { snapshot ->
                snapshot.reference.update( field: "spareChangeToSend", spares)
                Log.d( tag: "SpareActivity",  msg: "[sendSpares] Removed all sent spare change from current user's Firestore successfully")

            }
            if (tossed == true) {
                val builder = AlertDialog.Builder( context: this)
                builder.setTitle("Regarding your fine")
                builder.setMessage("You will still get fined 75% on any received spare change for tossing some spare change earlier today.
                builder.setNegativeButton( text: "OK") { _: DialogInterface, _: Int -> }
```

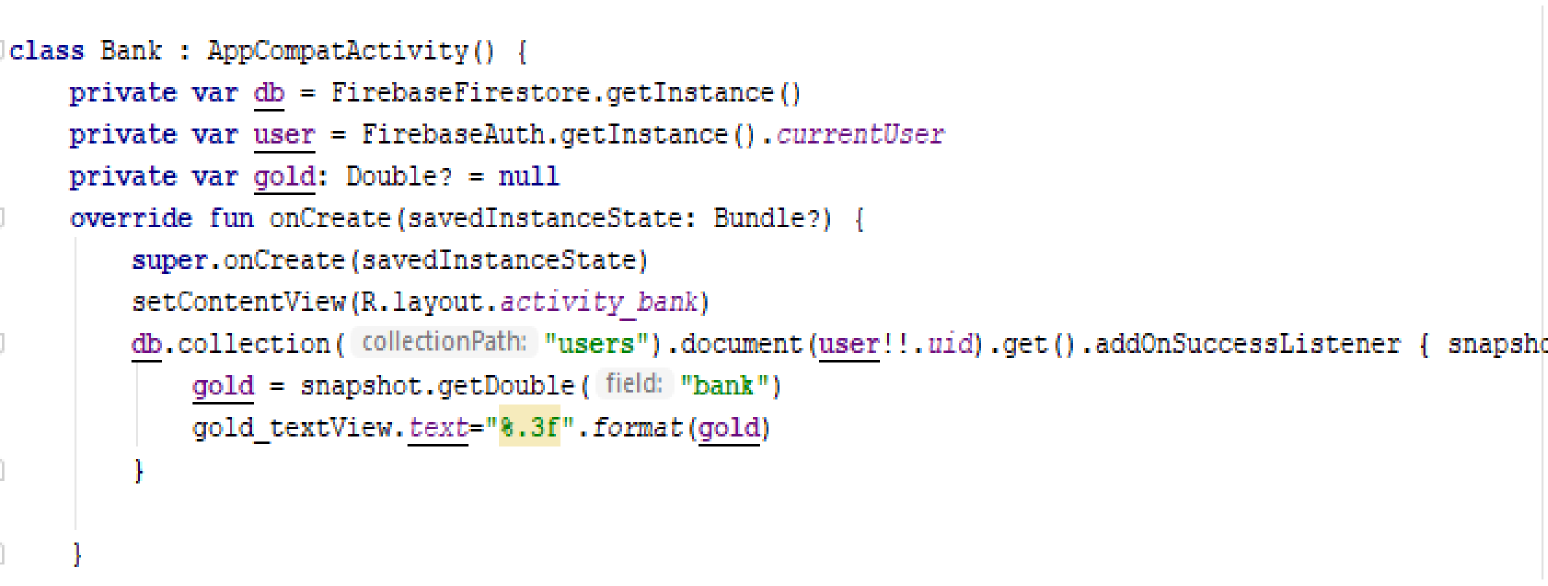
2.5. Bank Activity

The Bank Activity is responsible for displaying the current GOLD value that the user has in his/her account, which is rounded to 3 decimal places for readability.

The Activity can be accessed either via the Main Menu Activity, or via the App Bar (see *3.3 Extra Implementations*), located on the top of the screen in Classic Mode Activity.

The activity retrieves the *“bank”* field from FireStore (which is updated each new day in *Main Activity – see 2.2.2.)*, which contains the GOLD value of the current user and displays it (see *7.Screenshots)*.

A snippet showing the Bank Activity’s code is shown below:



```
class Bank : AppCompatActivity() {
    private var db = FirebaseFirestore.getInstance()
    private var user = FirebaseAuth.getInstance().currentUser
    private var gold: Double? = null
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_bank)
        db.collection( collectionPath: "users").document(user!!.uid).get().addOnSuccessListener { snapshot
            gold = snapshot.getDouble( field: "bank")
            gold_textView.text="%.3f".format(gold)
        }

    }
```

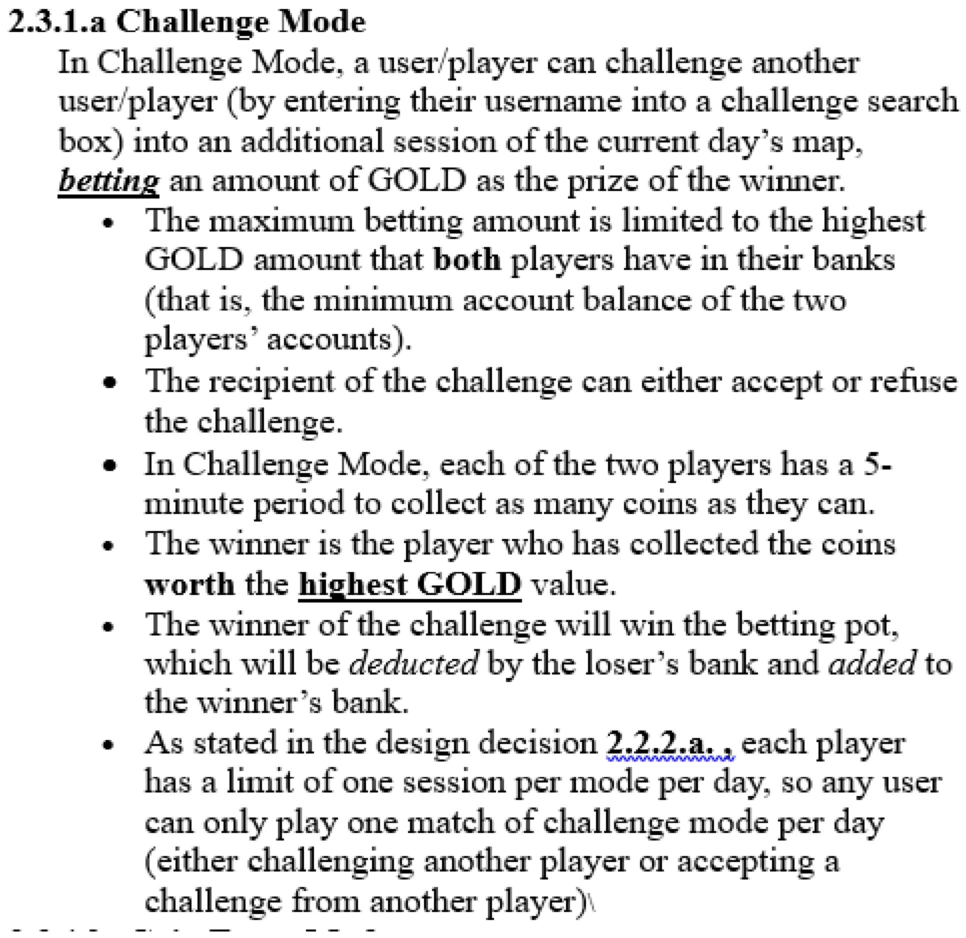
These 4 activities combined conclude the *core functionality* of the application. In the following two sections the *extra functionality* of the application is discussed.

3. Design Decisions and discrepancies from plan

This section discusses the differences between the implementation and the design decisions produced in Coursework 1. Particularly, the basic functionality was implemented *as designed*, while *one* design decision was implemented *differently* and *one bonus feature* was *not implemented at all.*

3.1. Designs that were not implemented

The only design produced in coursework 1 that was entirely removed from the implementation was the *Bonus Feature: Challenge Mode*. The following is a snippet of the Coursework 1 description of this bonus feature mode:



2.3.1.a Challenge Mode

In Challenge Mode, a user/player can challenge another user/player (by entering their username into a challenge search box) into an additional session of the current day's map, ***betting*** an amount of GOLD as the prize of the winner.

- The maximum betting amount is limited to the highest GOLD amount that **both** players have in their banks (that is, the minimum account balance of the two players' accounts).
- The recipient of the challenge can either accept or refuse the challenge.
- In Challenge Mode, each of the two players has a 5-minute period to collect as many coins as they can.
- The winner is the player who has collected the coins **worth** the **highest GOLD** value.
- The winner of the challenge will win the betting pot, which will be *deducted* by the loser's bank and *added* to the winner's bank.
- As stated in the design decision **2.2.2.a. ,** each player has a limit of one session per mode per day, so any user can only play one match of challenge mode per day (either challenging another player or accepting a challenge from another player)\

Reasons of removal:

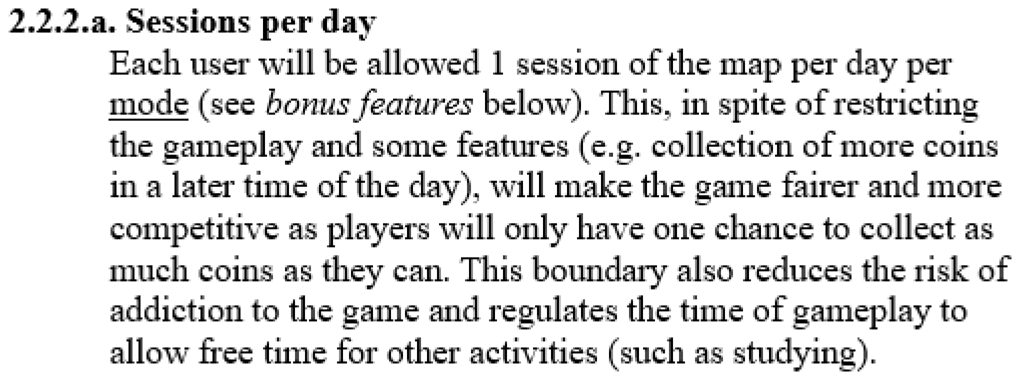
The main reason for not implementing this bonus feature was its *real-time requirements* of interfaces and interactions between *two different devices* (i.e. two users playing the game from different devices). This occurs when a user sends a challenge request to the other user, and the other user must accept the challenge before entering the challenge mode session.

Furthermore, the user who initiated the request has to remain idle and wait for the response of the other user. Moreover, there had to be some kind of push notification or interrupt signal to the other user's device to prompt him/her to accept/refuse the challenge, *wherever he is currently in the app.* Another problem raised is when the other user is *not inside the app* at the moment. This imposes the requirement of *online status.*

The above reasons have resulted in me *removing this feature from implementation.*

3.2. Designs that were implemented differently (discrepancies)

There was *one* design decision that was implemented *differently* than the Coursework 1 design specification. The design decision was the *one session per day limit.*
The following is a snippet from the design decision as documented in Coursework 1:



2.2.2.a. Sessions per day

Each user will be allowed 1 session of the map per day per mode (see *bonus features* below). This, in spite of restricting the gameplay and some features (e.g. collection of more coins in a later time of the day), will make the game fairer and more competitive as players will only have one chance to collect as much coins as they can. This boundary also reduces the risk of addiction to the game and regulates the time of gameplay to allow free time for other activities (such as studying).

This feature was sensible to impose not only for the reasons described in the snippet above, but also to raise fair competitiveness as the players will only be allowed to play game modes once per day, thus disallowing some players spending more time on the game to play multiple times to be ranked better than the rest of the players.
However, upon implementation of the *Classic Mode Activity*, I realised that the map covers a big area and 50 coins spread out in the area of the campus are a lot to collect in one single session. Also, if the user accidentally exited the Classic Mode, then all potential progress is lost. Hence, I decided to *allow multiple sessions for Classic Mode,* but only *apply the limit to Coin Fever Mode.*
This was a perfectly sensible decision as in Coin Fever Mode the user plays in a new instance of the map, attempting to collect the same day's coins *again* by running against the 3-minute clock (see section *4.1.*). If multiple sessions of Coin Fever Mode were allowed, then one could have a significant advantage whilst collecting more and more coins in each session of the mode.

3.3. Extra Implementations

The following list are some *extra implementations* that were not included in the Coursework 1 specification document:

- **Legend of rates**: When the user is playing Classic Mode, he is interested in knowing the currency rates of coins *on the spot*, in order to collect the coins that value the most (for the first 25 coins which are cashed in to the user's bank account). Thus, I implemented a legend showing the currency rates for each currency, along with the coin icon that matches that currency.
  The legend of rates can be seen in section *7. Screenshots* in the screenshots of Classic Mode.
- **ActionBar/App Bar**: I implemented an ActionBar (App Bar) on the top edge of Classic Mode Activity, so that the user can quickly and conveniently navigate to the Spare Change Activity and Bank Activity, as well as selecting the currencies to view/hide their coins from the map (see below bullet point)
  The action bar can be viewed in section *7. Screenshots* in the screenshots of Classic Mode.

- **ActionMenu** for Showing/Hiding specific currency coins (see 4.2.):
  I implemented an *action menu* (instead of buttons) for showing/hiding coins of a currency from the map in Classic Mode. I placed the Action Menu on the *App Bar* (see above bullet point). The action menu can be viewed in section *7. Screenshots* in the screenshots of Bonus Features(showing/hiding currencies).

4. Bonus Features

The following sub-sections describe the bonus features that I implemented as per the bonus features proposed in Coursework 1's design specification. The only bonus feature that was *not* implemented was Challenge Mode (see *3.1.)*

4.1. Coin Fever Mode

The biggest bonus feature that I implemented was the Coin Fever Mode. In this mode, the user can play in a new map of the same day (all coins present) and run against a 3-minute countdown timer to collect 10 or more coins.

If the user collects 10 or more coins, the coins are stored in a **HashMap** variable *feverModeCollectedCoinz* similar to the *Classic Mode* variables, and converted to GOLD the next day (in *Main Activity, see 2.2.2).*

Of course, when the timer reaches zero, a *flag variable* is set to *true* to restrict the user from collecting any more coins after the timer is finished.

The Activity displays different alert dialogs to inform the user of how many coins he/she collected (also shown in bottom left corner of screen) and whether he/she has reached 10 or more coins or not.

The user then presses the button on the dialog and is redirected back to *Main Activity,* where now the Coin Fever Mode button does *not* allow the user to play another session in the same day, informing him/her of the daily limit with an alert dialog (see *7. Screenshots* under *Coin Fever Mode*).
The (well documented) code for the Coin Fever Mode Activity can be reviewed in the Android Studio Project files.

4.2. Show/Hide specific currency of coins on map

This bonus feature allows the user/player to hide or show a chosen currency's coins on the map. (see *7. Screenshots* under *Bonus Features-Hide currencies*). This is to help the player focus on a specific group of target coins of a particular currency, which is extremely helpful combined with the *currency rates legend (see 3.3. Extra Implementations)* for the user to know which currency is the most valuable on the spot.
The user can choose which currency to display its coins on the map using an *action menu (see 3.3),* and, of course, the user will *only be able to collect the visible coins on the map.* When the user chooses the View All Coins option, then all remaining coins (collected coins are removed) are re-displayed on the map.

4.3. Fair Play Fine

This bonus feature encourages fair play between players, in the sense that players who *do not* send spare coins to their friends should have a penalty. (see *7. Screenshots* under *Bonus Features-Fair Play Fine*)

If a player collects more than 25 coins in Classic Mode (the original game as specified in the coursework) and decides to *toss his spare change* rather than sending it to another player, then the extra coins are tossed and in the case that he/she receives spare change from another player, a 75% fine will be applied on any amount of spare change received.

The flag that indicates that the player has tossed the spare change and the fine must be applied to him happens in *Spare Change Activity* (see *7. Screenshots* under *Bonus Features-Fair Play Fine*).

The calculation of the fine on the *receivedSpares* GOLD value happens inside *Main Activity (see 2.2.2.)*

4.4. Leaderboard

A player leader board Activity showing the player names (usernames) sorted in descending order of GOLD in bank (using a ListView). The current user's entry in the leaderboard is indicated with a "(You)" next to the e-mail address.

This bonus feature encourages competitiveness among players, as it introduces the concept of ranks among players with the most GOLD value.

See *7. Screenshots* under *Bonus Features-Leaderboard.*

5. Testing

Apart from manual testing of the app, I performed some *instrumented tests* using espresso, as instructed in the coursework specification. Because of the limitation of espresso tests to only interface *assertions*, I could not test every single feature of my application. The following list includes a description for each test name implemented.
Note: On testing time (complete code of application), all tests ran to completion. Sometimes espresso does not run smoothly and causes problems with the test.

- SignInOutTest: In this test I perform a simple sign in (assume clean install of the app) and sign out of the user test@gmail.com (password:test1234).
- RegisterTest: In this test I attempt a register with the *same* e-mail and password, and check for the appropriate Toast message by using my Toast Matcher class (see 6. Acknowledgements)
- ClassicModeTest: In this test I sign-in, enter Classic Mode and exit again to sign-out (in order to maintain order-free tests), checking that I entered the mode by asserting an icon on the App Bar (see 3.3)

In manual testing I used the following virtual and physical devices:

- Nexus 5X emulator with Android 8.0 (Oreo API 26)
- OnePlus 3T (also running API 26)

For further testing, the markers can feel free to either use the test user (either *test* or *test2@gmail.com* with password test1234) or register a new user.

6. Acknowledgements and References to sample code us

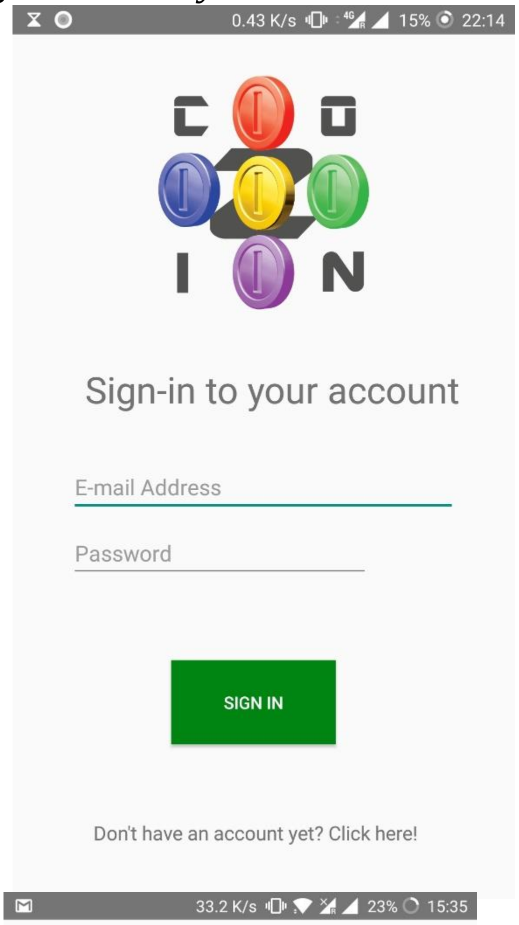
Apart from the sample code included in the lectures, I used some additional online resources in parts of my implementation. The following list contains links for each part of code which I used sample code or guidance from online resources:

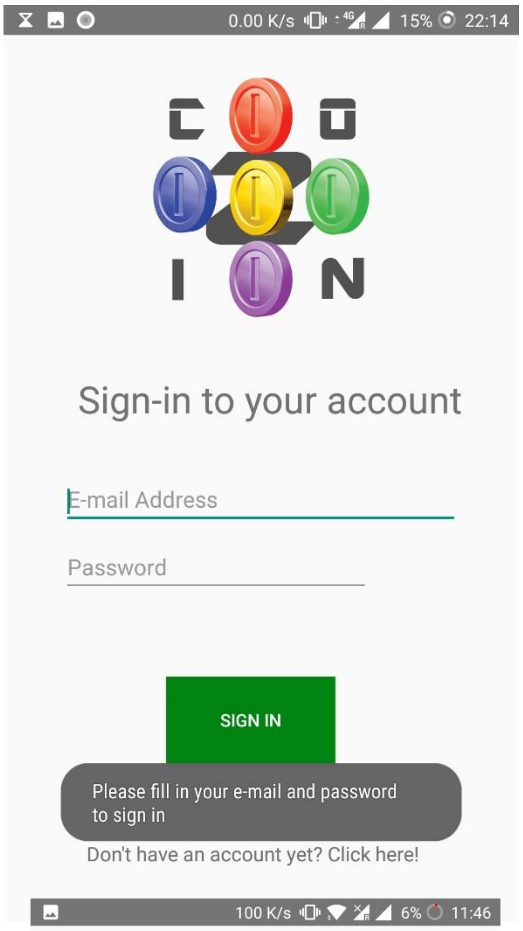
- Sign in and Register Activities
- FireStore access and queries
- App Bar
- Countdown Timer
- Countdown Timer (2)
- Action Menu
- Action Menu (2)
- Leaderboard
- Leaderboard (2)
- Toast Matcher (for Testing)

7. Screenshots

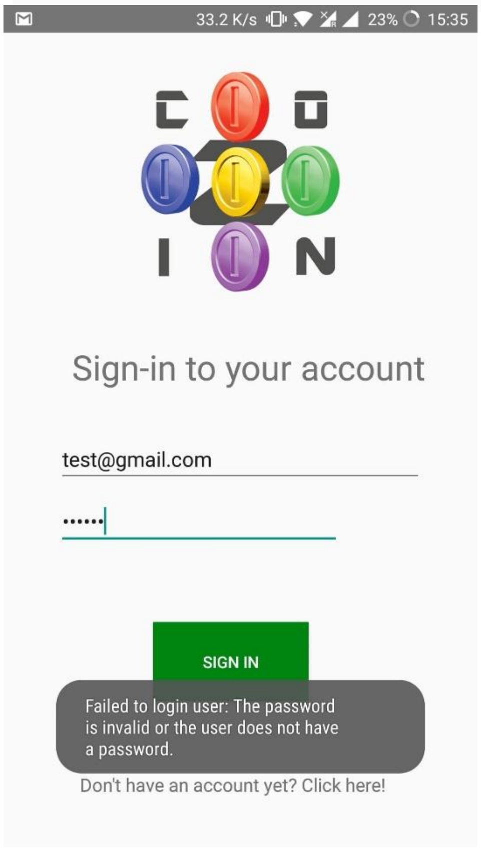
The following pages contain screenshots of the application in-use, showing its different features and cases of gameplay.
Some error checking screenshots (with toast messages) are also included.
The screenshots were taken on my physical OnePlus 3T device running Android 8.0 (Oreo API 26)

Sign In Activity

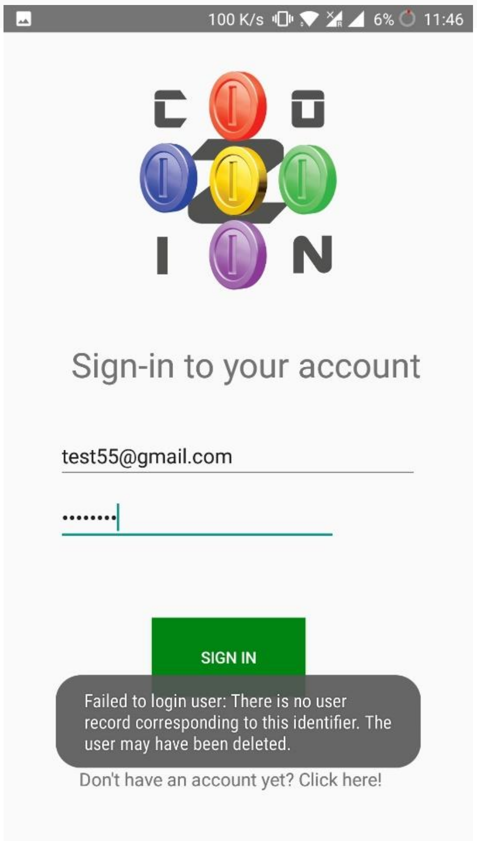




Please fill in your e-mail and password to sign in

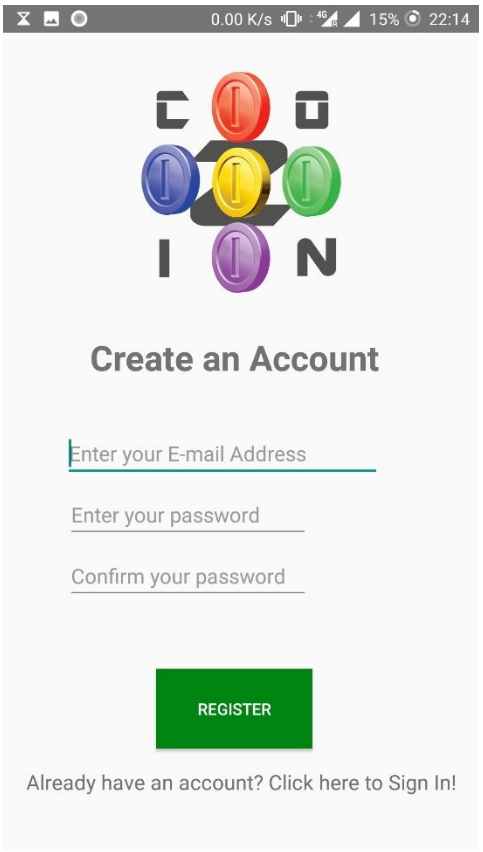


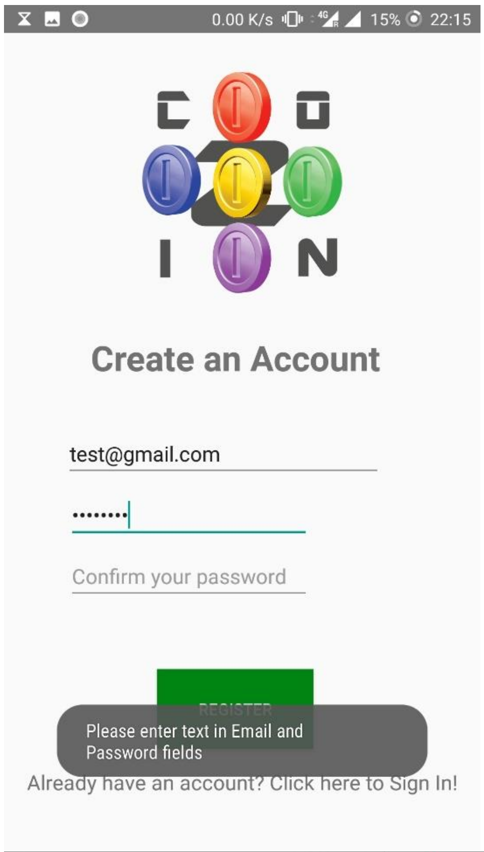
Failed to login user: The password is invalid or the user does not have a password.



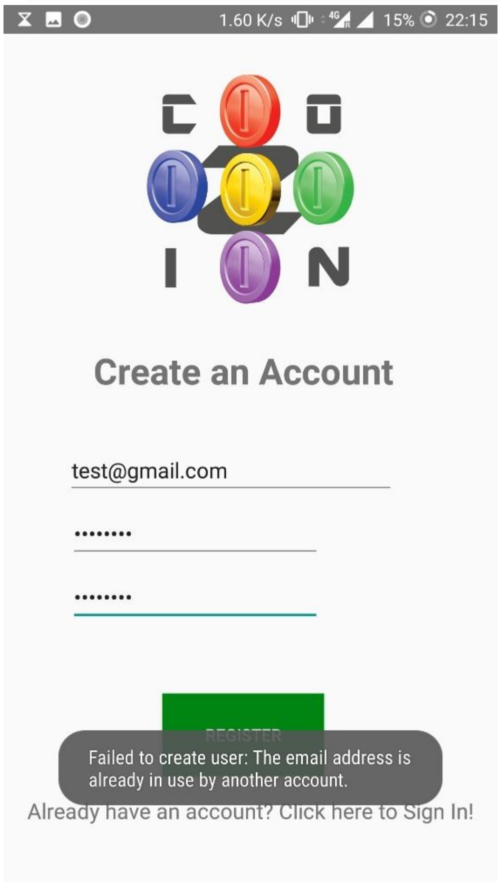
Failed to login user: There is no user record corresponding to this identifier. The user may have been deleted.

Register Activity

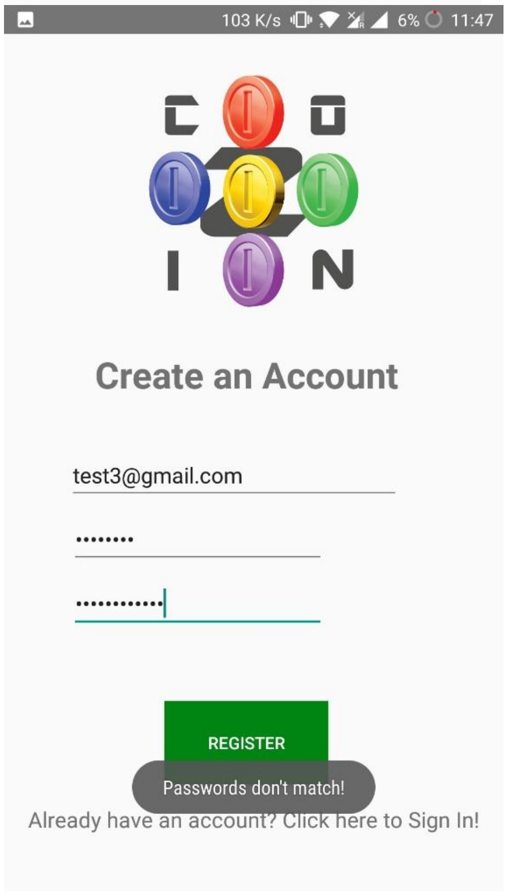




Please enter text in Email and Password fields



Failed to create user: The email address is already in use by another account.



Passwords don't match!

Main Activity

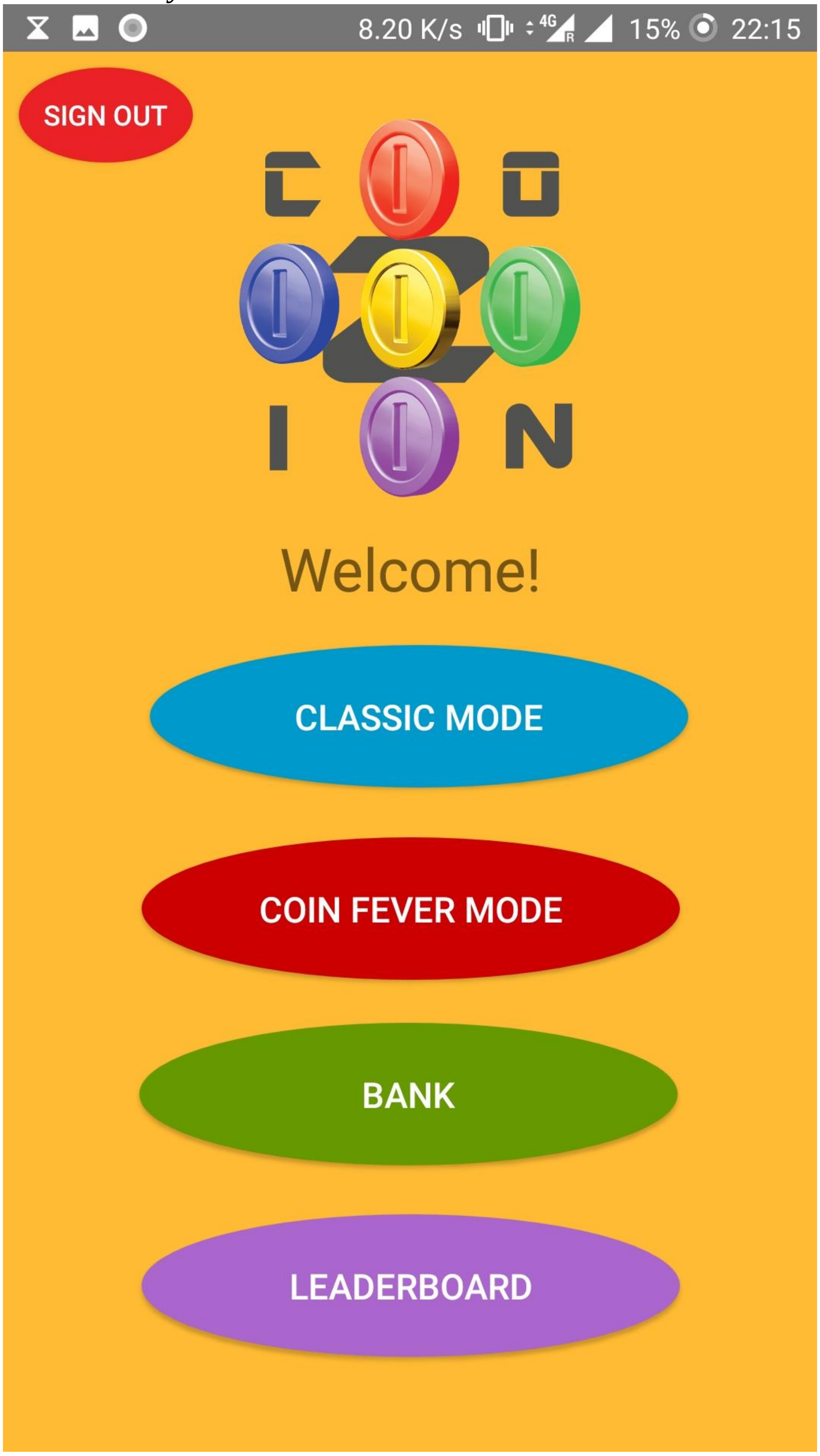
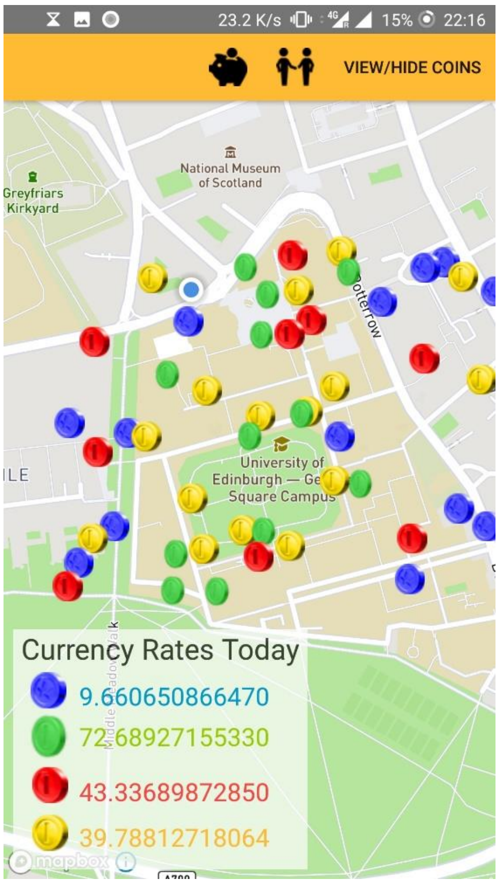
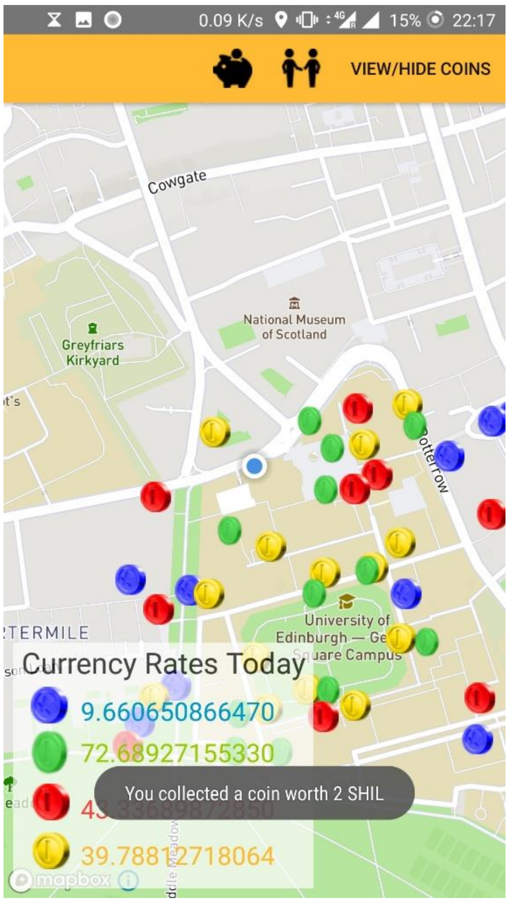


Figure 1 Main Activity: DownloadTask of GeoJSON file in the background, conversion of previous day coins to GOLD and deposit into bank account of user, parsing of today's currency rates

Classic Mode Activity


23.2 K/s 15% 22:16
VIEW/HIDE COINS
National Museum of Scotland
Greyfriars Kirkyard
Potterrow
University of Edinburgh — George Square Campus
mapbox

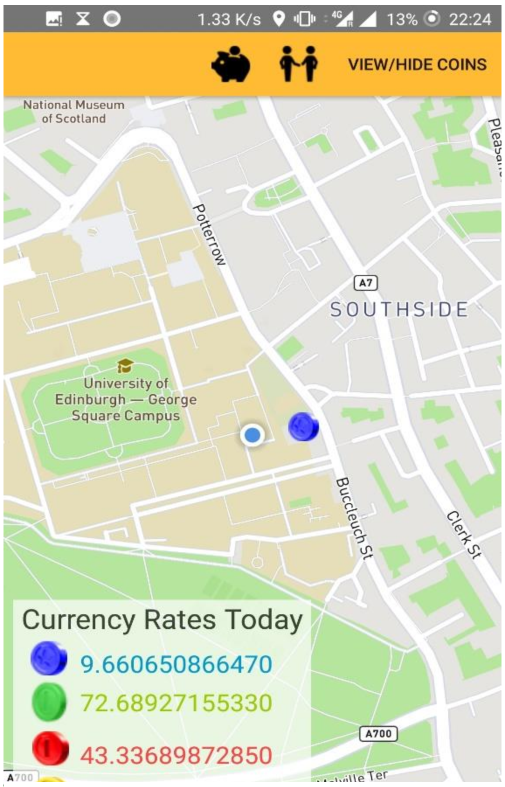
Currency Rates Today
9.660650866470
72.68927155330
43.33689872850
39.78812718064



Currency Rates Today
9.660650866470
72.68927155330
43.33689872850
39.78812718064

You collected a coin worth 2 SHIL

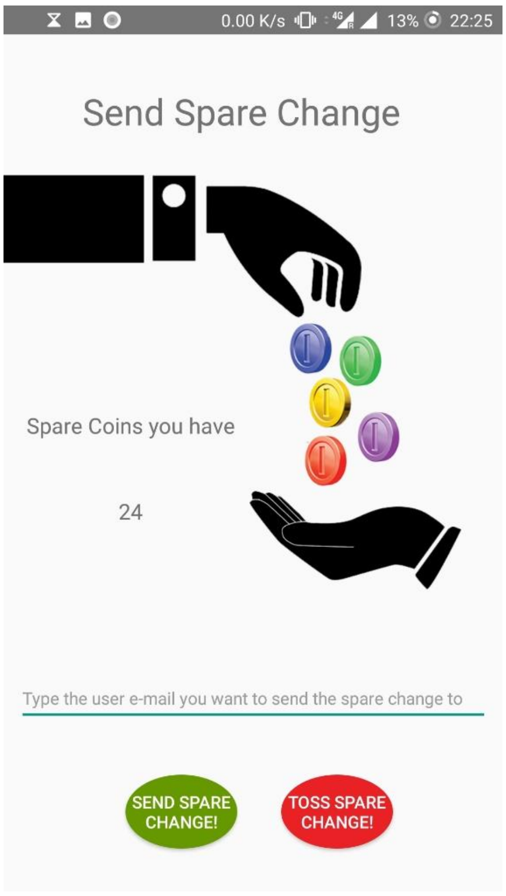
Figure 3 Collecting a coin (automatic collection within radius of 25m)

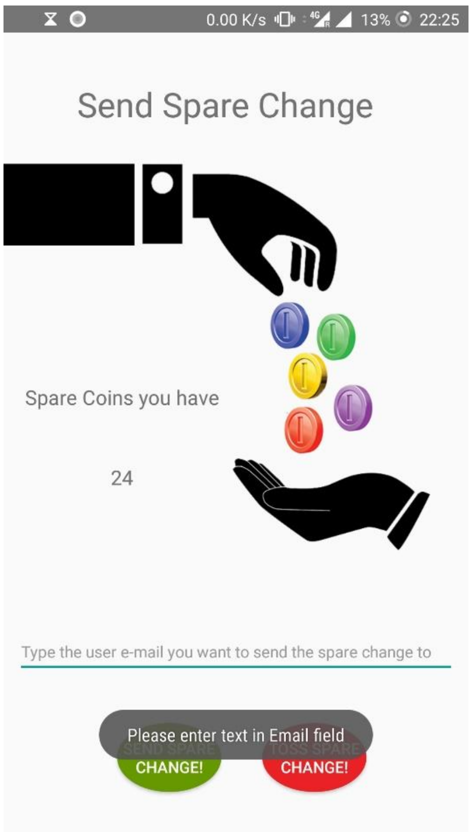


Currency Rates Today
9.660650866470
72.68927155330
43.33689872850

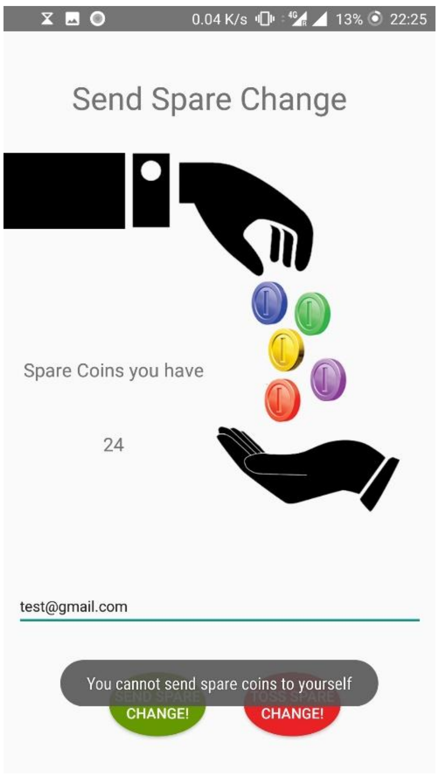
Figure 3 Collected 49 coins

Spare Change Activity



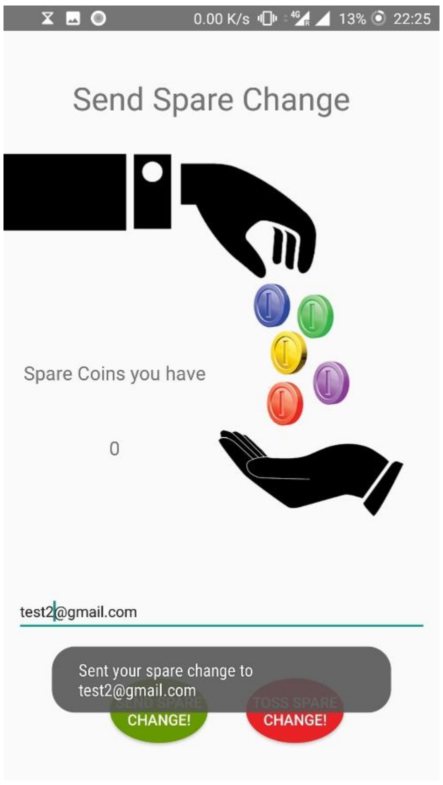


Please enter text in Email field



test@gmail.com

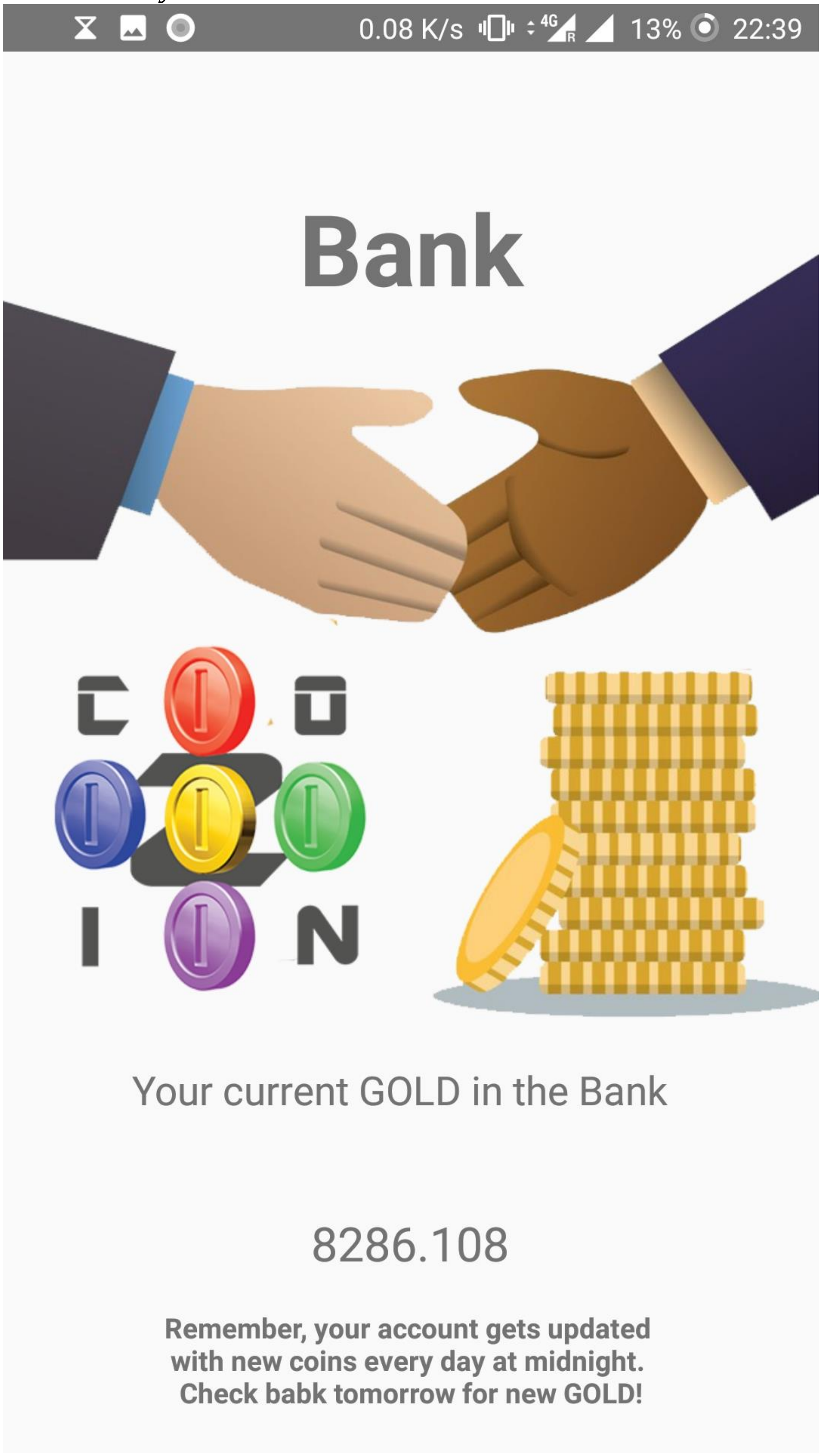
You cannot send spare coins to yourself



test2@gmail.com

Sent your spare change to test2@gmail.com

Bank Activity



0.08 K/s 13% 22:39

Bank

Your current GOLD in the Bank

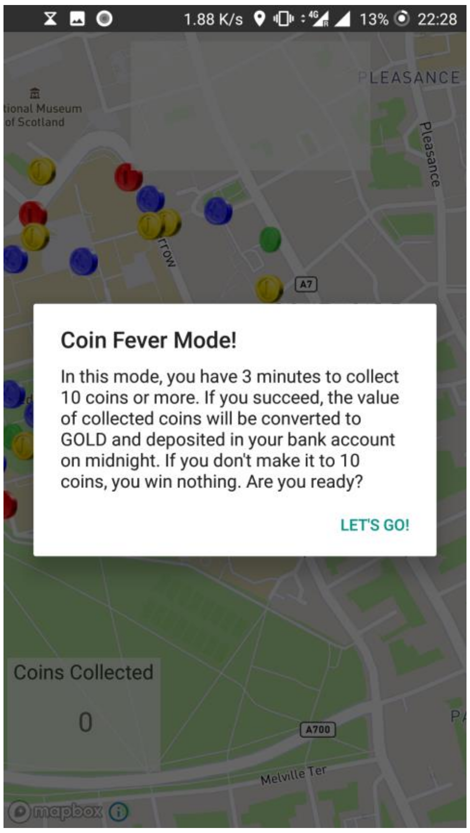
8286.108

Remember, your account gets updated
with new coins every day at midnight.
Check babk tomorrow for new GOLD!

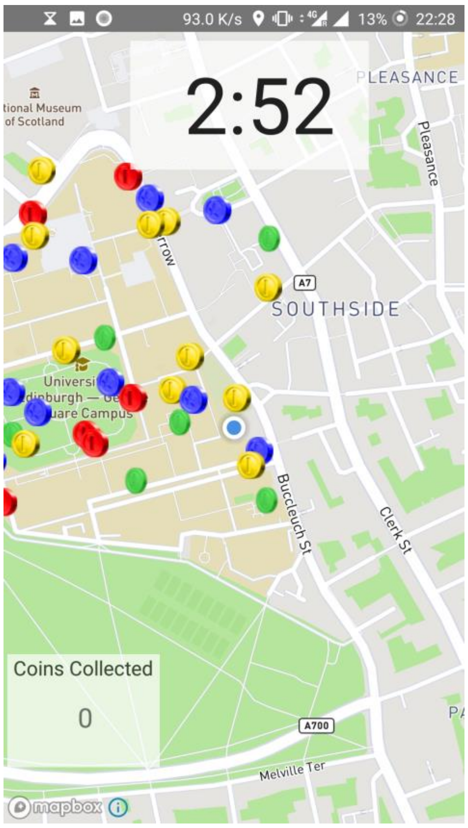
Figure 4 GOLD Balance is shown to the user in 3 decimal places

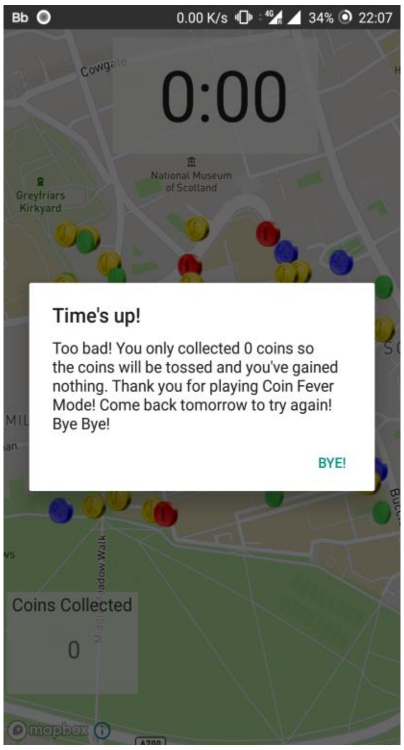
BONUS FEATURES

Coin Fever Mode

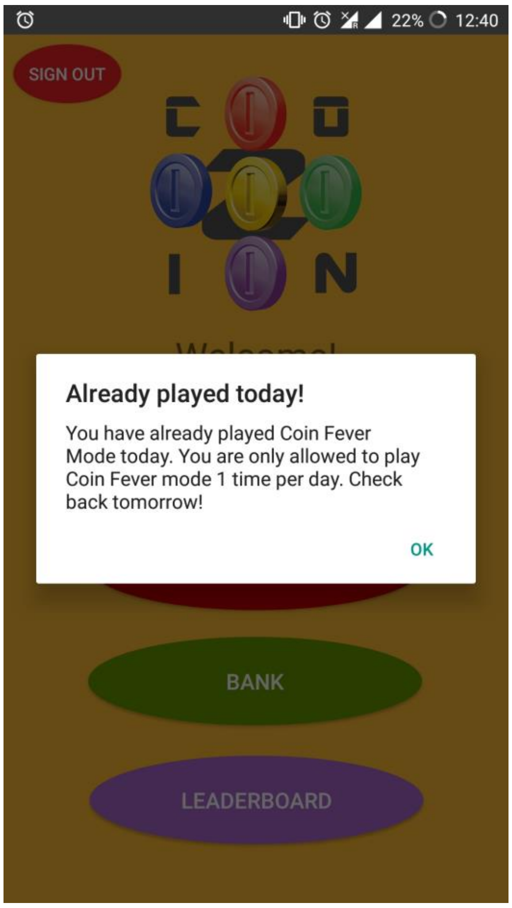

Coin Fever Mode!

In this mode, you have 3 minutes to collect 10 coins or more. If you succeed, the value of collected coins will be converted to GOLD and deposited in your bank account on midnight. If you don't make it to 10 coins, you win nothing. Are you ready?

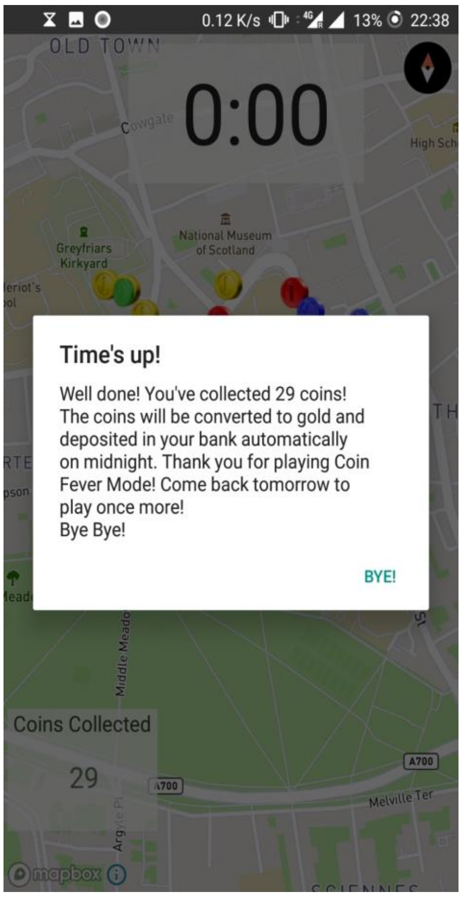



Time's up!

Too bad! You only collected 0 coins so the coins will be tossed and you've gained nothing. Thank you for playing Coin Fever Mode! Come back tomorrow to try again! Bye Bye!

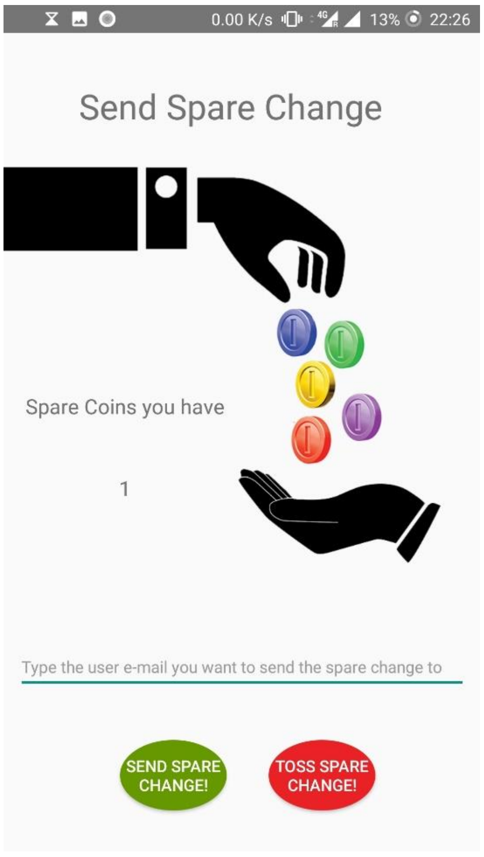

Already played today!

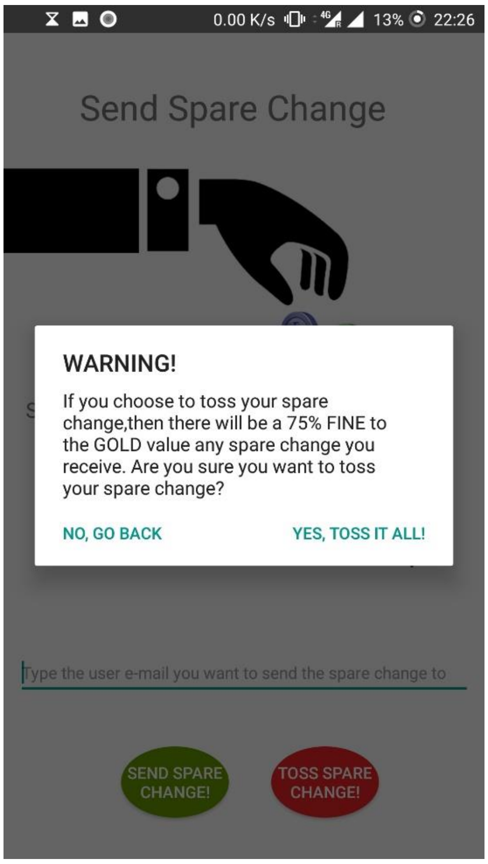
You have already played Coin Fever Mode today. You are only allowed to play Coin Fever mode 1 time per day. Check back tomorrow!


Time's up!

Well done! You've collected 29 coins! The coins will be converted to gold and deposited in your bank automatically on midnight. Thank you for playing Coin Fever Mode! Come back tomorrow to play once more!
Bye Bye!

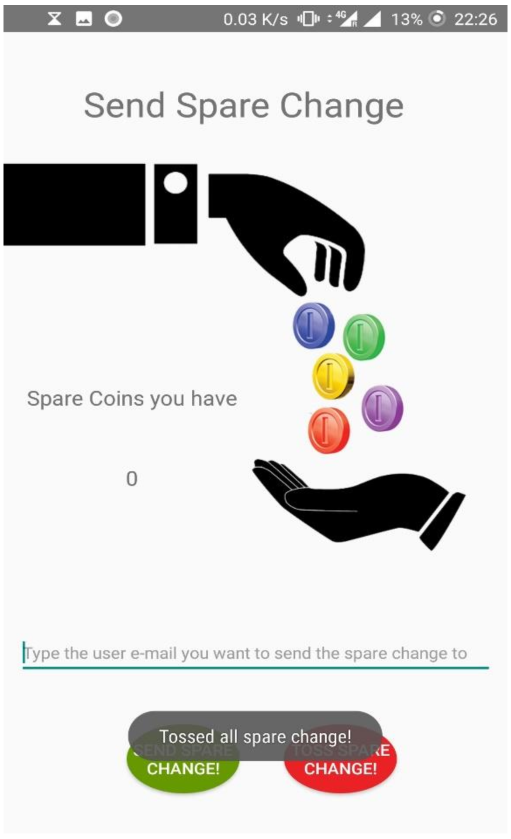
Fair Play Fine



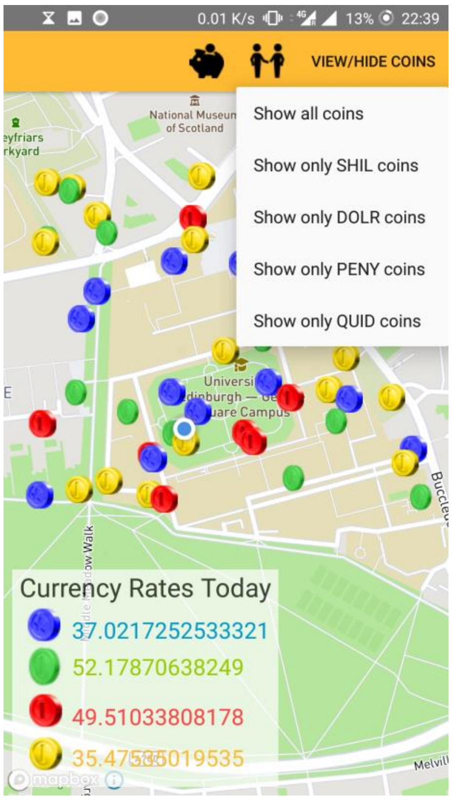


WARNING!

If you choose to toss your spare change,then there will be a 75% FINE to the GOLD value any spare change you receive. Are you sure you want to toss your spare change?

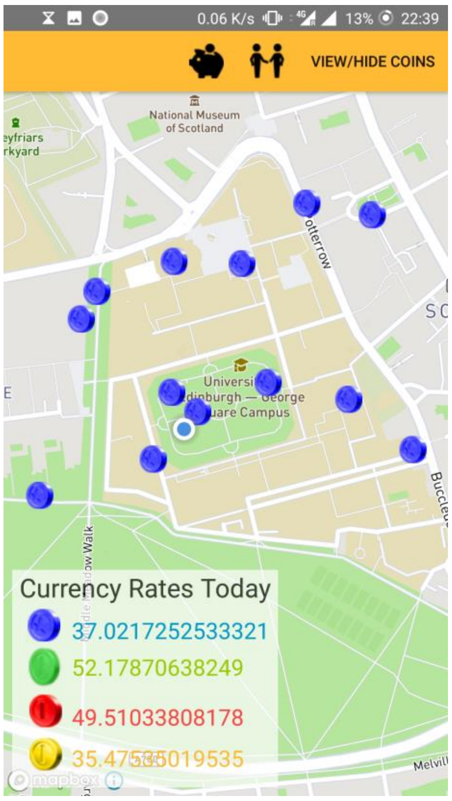


Hide Currencies

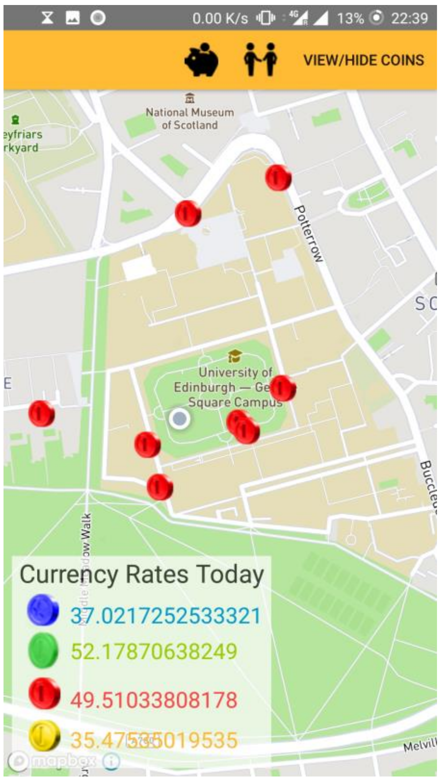

0.01 K/s 13% 22:39
VIEW/HIDE COINS
National Museum of Scotland
Greyfriars Kirkyard
University of Edinburgh — George Square Campus
Buccleuch

Show all coins
Show only SHIL coins
Show only DOLR coins
Show only PENY coins
Show only QUID coins

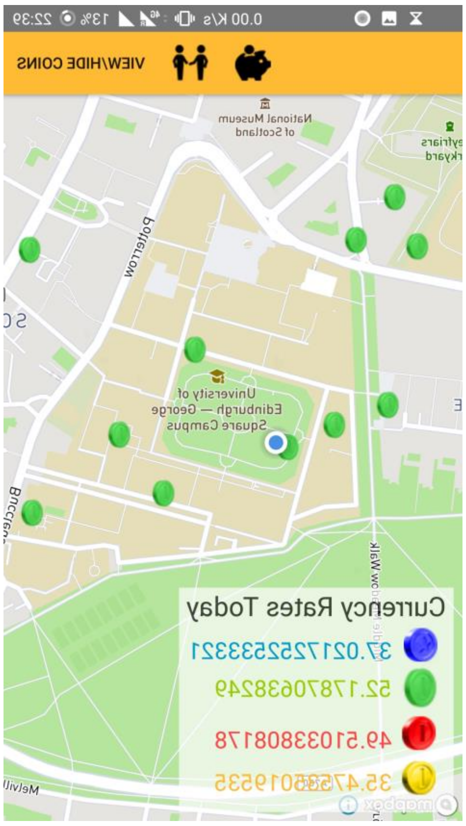
Currency Rates Today
37.0217252533321
52.17870638249
49.51033808178
35.47585019535


0.06 K/s 13% 22:39
VIEW/HIDE COINS
National Museum of Scotland
Potterrow
University of Edinburgh — George Square Campus

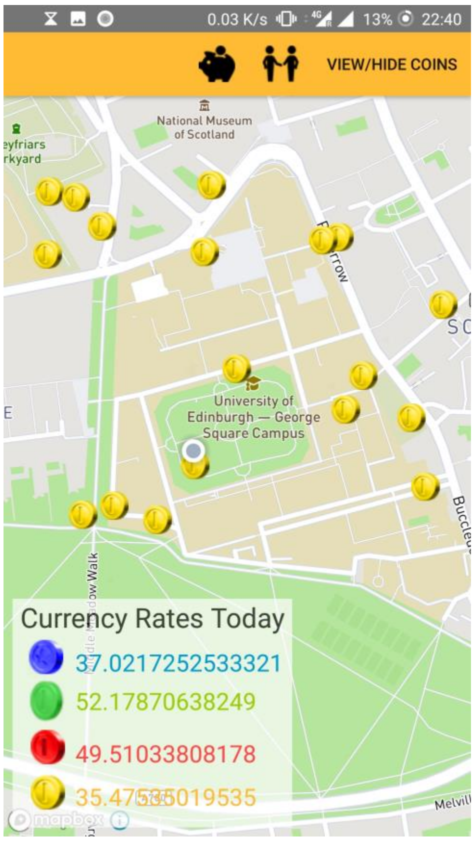
Currency Rates Today
37.0217252533321
52.17870638249
49.51033808178
35.47585019535


0.00 K/s 13% 22:39
VIEW/HIDE COINS
National Museum of Scotland
Potterrow
University of Edinburgh — George Square Campus

Currency Rates Today
37.0217252533321
52.17870638249
49.51033808178
35.47585019535

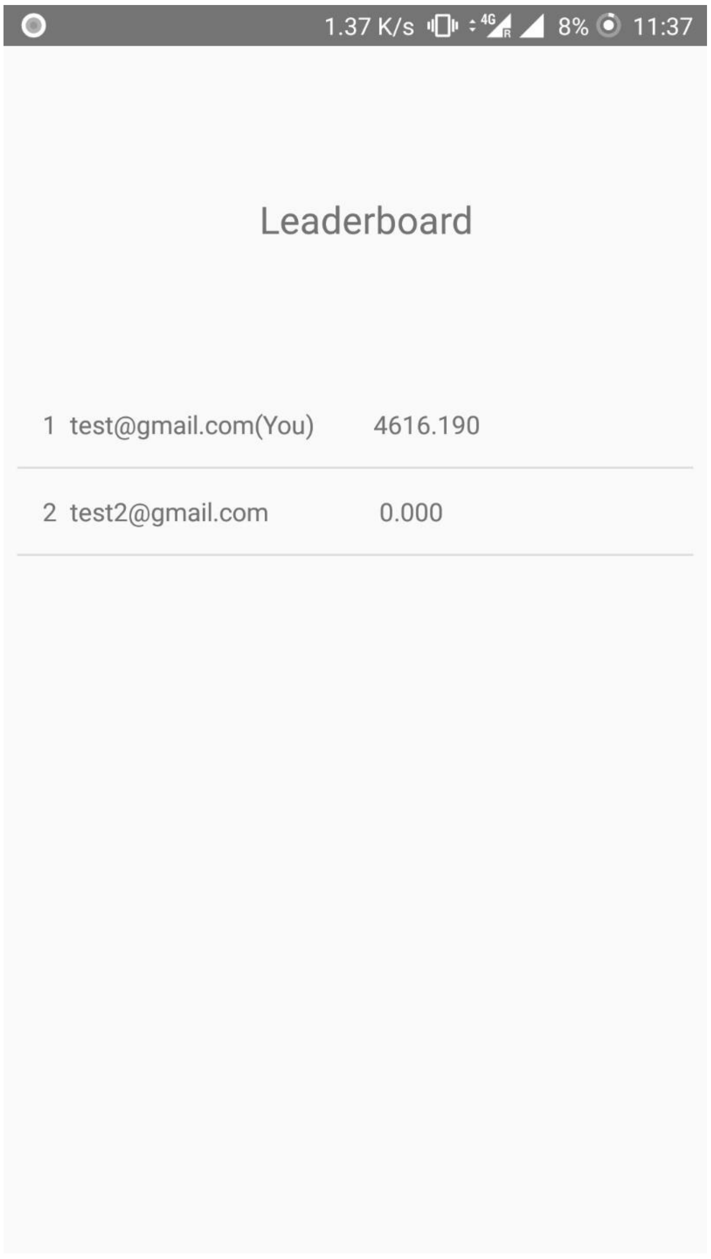

0.00 K/s 13% 22:39
VIEW/HIDE COINS
National Museum of Scotland
Potterrow
University of Edinburgh — George Square Campus

Currency Rates Today
37.0217252533321
52.17870638249
49.51033808178
35.47585019535


0.03 K/s 13% 22:40
VIEW/HIDE COINS
National Museum of Scotland
University of Edinburgh — George Square Campus
Buccleuch

Currency Rates Today
37.0217252533321
52.17870638249
49.51033808178
35.47585019535

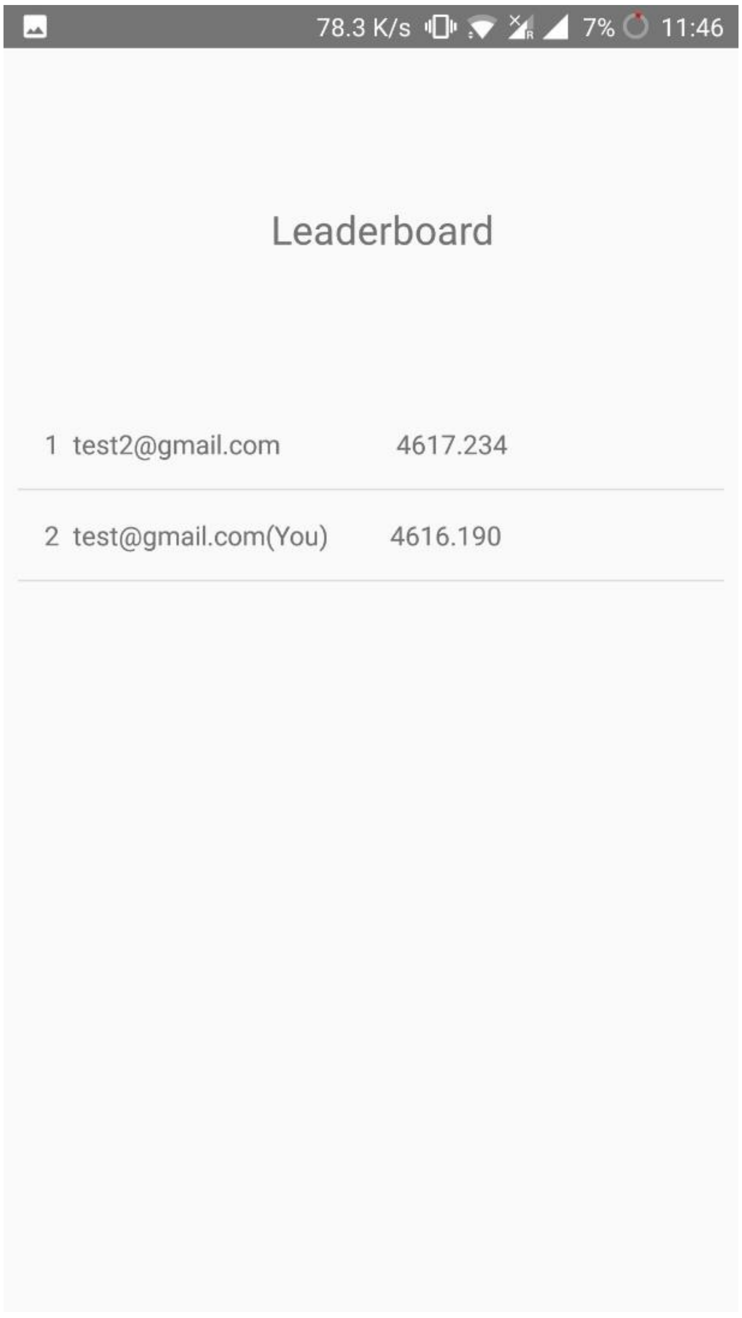
Leaderboard



1.37 K/s 8% 11:37

Leaderboard

1	test@gmail.com(You)	4616.190
2	test2@gmail.com	0.000



78.3 K/s 7% 11:46

Leaderboard

1	test2@gmail.com	4617.234
2	test@gmail.com(You)	4616.190