Final Project Report for Even Semester 2020

Submitted as course project of

OBJECT ORIENTED PROGRAMMING

COMP6699

Written by:

Karel Bondan Andoro Herdito - 2440032373

Under the Guidance of:

JUDE MARTINEZ

Lecturer of Object-Oriented Programming

School of Computer Science

Binus International University

Project Requirements

Functional Requirements

Specifically, the final project application MUST include examples of the following AT A MINIMUM:

- Use of primitive data
- Use of instance variables and objects
- Use of imported classes
- Use of custom-built classes & methods
- Use of Java Collection
- Use of exception handling
- Use of inheritance, polymorphism and interfaces
- Detailed Documentation Commenting
- Detailed Commenting of Methods
- Detailed Code Commenting

Non-Functional Requirements

Creating a working implementation is far from sufficient to achieve full marks!

- Stick strictly to the above specifications
- Don’t forget to add helpful comments
- Use (long) meaningful identifiers
- Use ample white space and a consistent indentation scheme
- Make code easy to read by keeping it simple
- Avoid duplicating similar code
- Ensure that all domain classes are modeled properly
- Use a proper access control to variables and methods

Plagiarism/Cheating

Binus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept and consent to Binus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:

Karel Bondan Andoro Herdito

I. Project Background

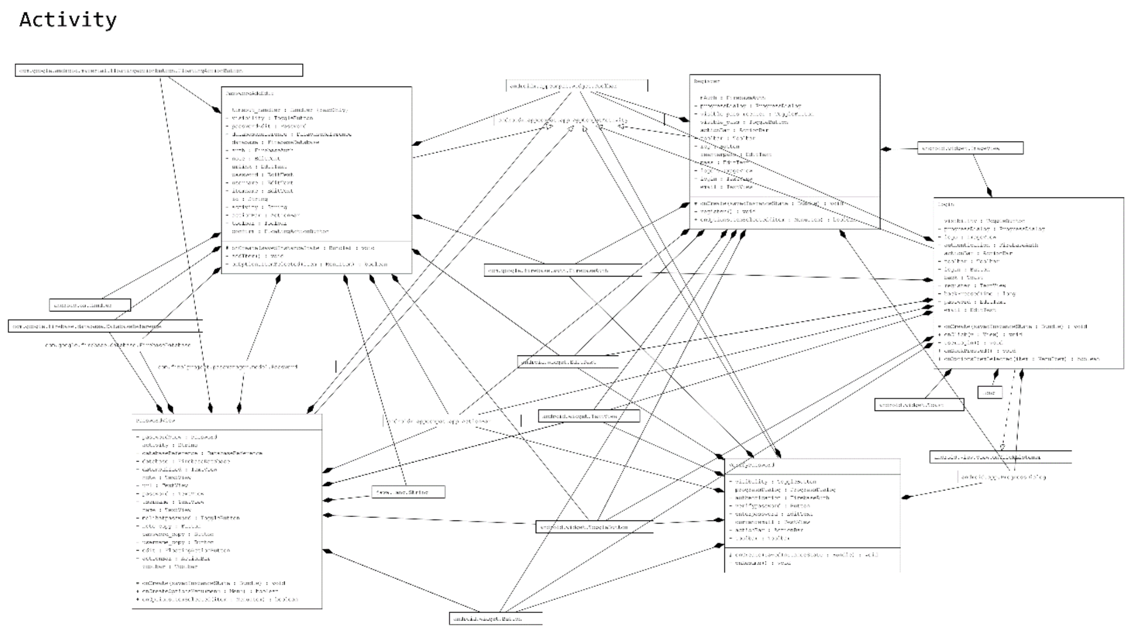
I made an GUI application called “Password Notes”. It is an Android application that can save and store your password into the cloud so you can access it from anywhere, anytime you want.

II. Project Specifications

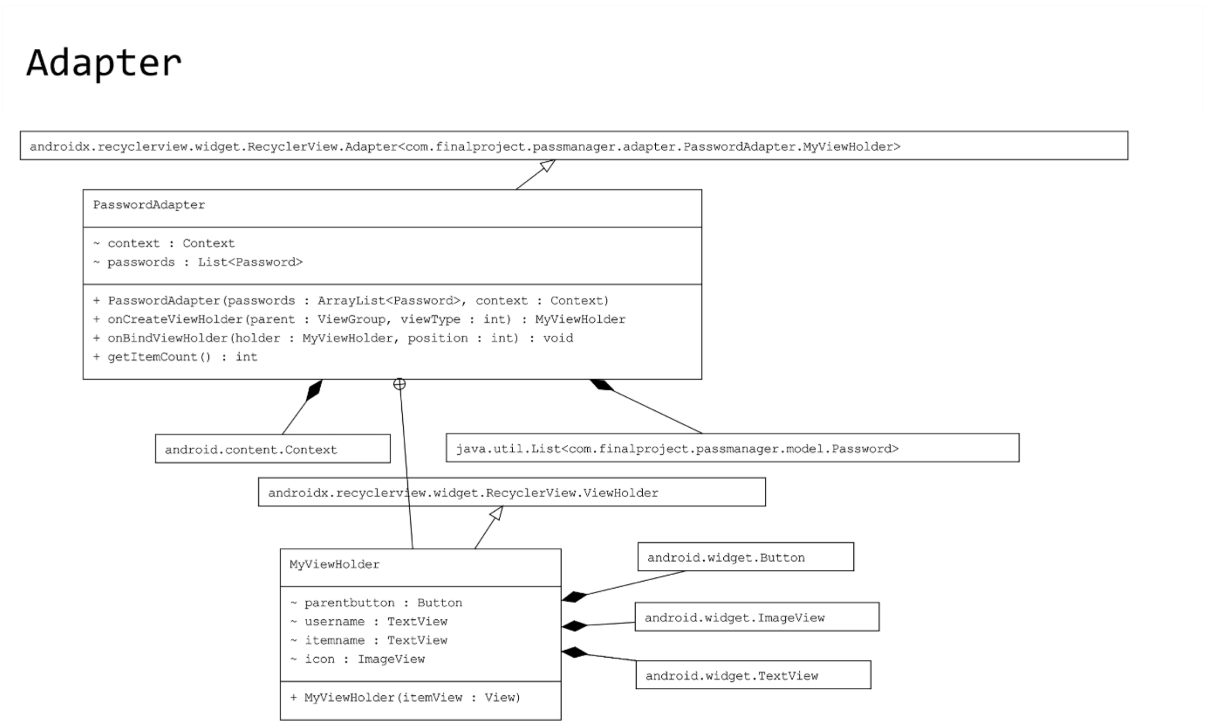
Password Notes is made using Android Studio, a Java compiler which is based on IntelliJ IDEA developed by JetBrains. It uses Android as the main operating system for the base application with user interactions to work. The cloud platform, which used to store the passwords uses Firebase as the main database.

III. UML Diagram

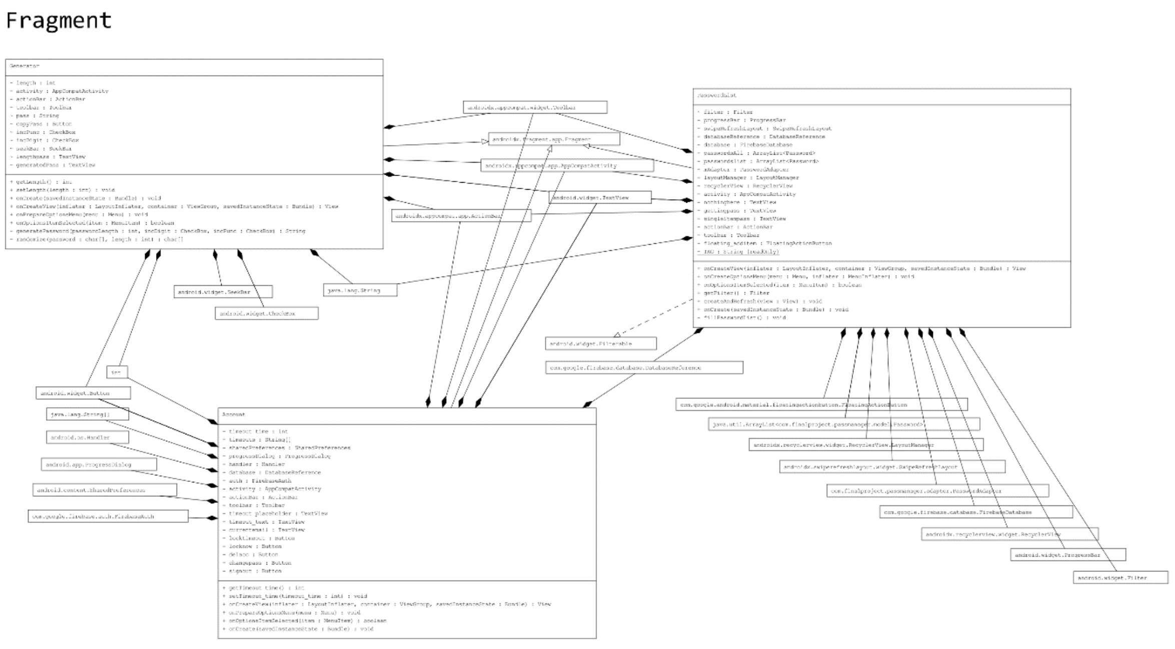
- **Activity package**



- **Adapter package**

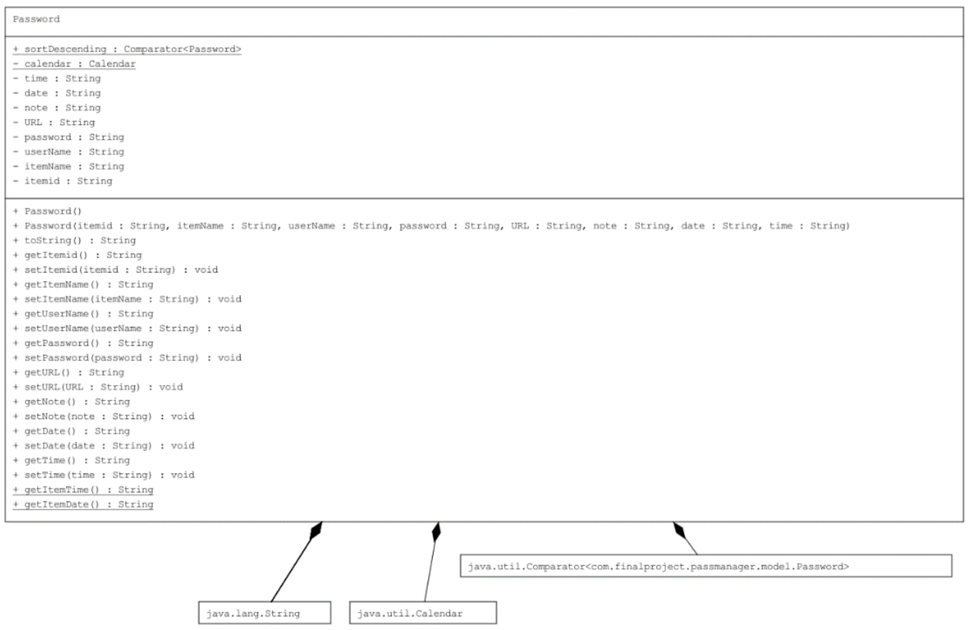

Adapter
androidx.recyclerview.widget.RecyclerView.Adapter<com.finalproject.passmanager.adapter.PasswordAdapter.MyViewHolder>
PasswordAdapter
~ context : Context
~ passwords : List<Password>
+ PasswordAdapter(passwords : ArrayList<Password>, context : Context)
+ onCreateViewHolder(parent : ViewGroup, viewType : int) : MyViewHolder
+ onBindViewHolder(holder : MyViewHolder, position : int) : void
+ getItemCount() : int
android.content.Context
java.util.List<com.finalproject.passmanager.model.Password>
androidx.recyclerview.widget.RecyclerView.ViewHolder
MyViewHolder
~ parentbutton : Button
~ username : TextView
~ itemname : TextView
~ icon : ImageView
+ MyViewHolder(itemView : View)
android.widget.Button
android.widget.ImageView
android.widget.TextView

- **Fragment package**



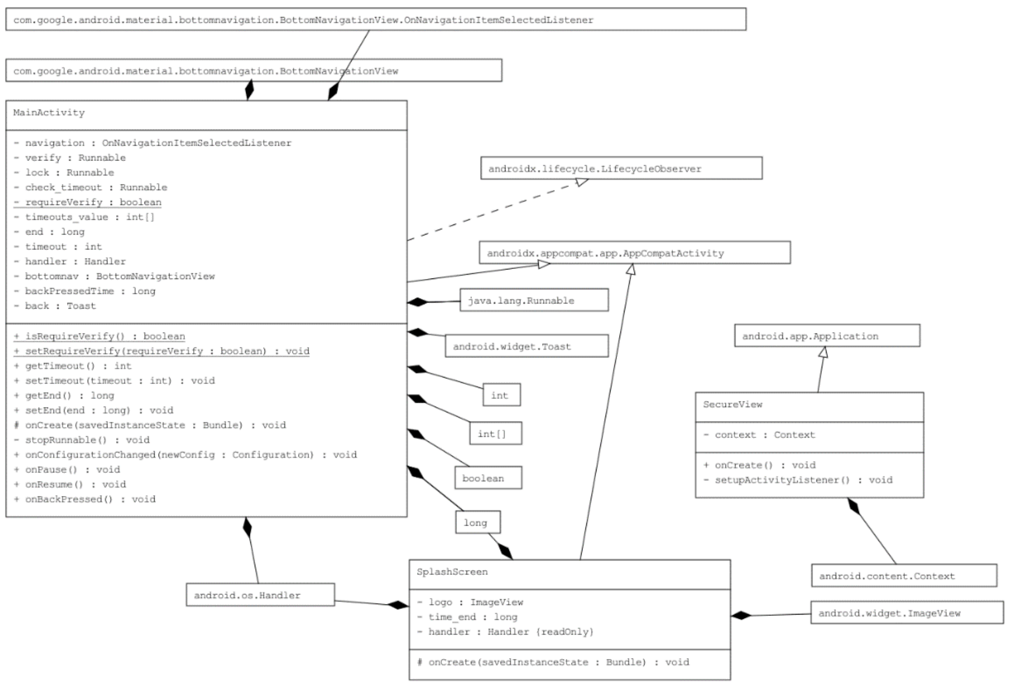
- **Model package**

Model


Password
+ sortDescending : Comparator<Password>
- calendar : Calendar
- time : String
- date : String
- note : String
- URL : String
- password : String
- userName : String
- itemName : String
- itemid : String
+ Password()
+ Password(itemid : String, itemName : String, userName : String, password : String, URL : String, note : String, date : String, time : String)
+ toString() : String
+ getItemid() : String
+ setItemid(itemid : String) : void
+ getItemName() : String
+ setItemName(itemName : String) : void
+ getUserName() : String
+ setUserName(userName : String) : void
+ getPassword() : String
+ setPassword(password : String) : void
+ getURL() : String
+ setURL(URL : String) : void
+ getNote() : String
+ setNote(note : String) : void
+ getDate() : String
+ setDate(date : String) : void
+ getTime() : String
+ setTime(time : String) : void
+ getItemTime() : String
+ getItemDate() : String
java.util.Comparator<com.finalproject.passmanager.model.Password>
java.lang.String
java.util.Calendar

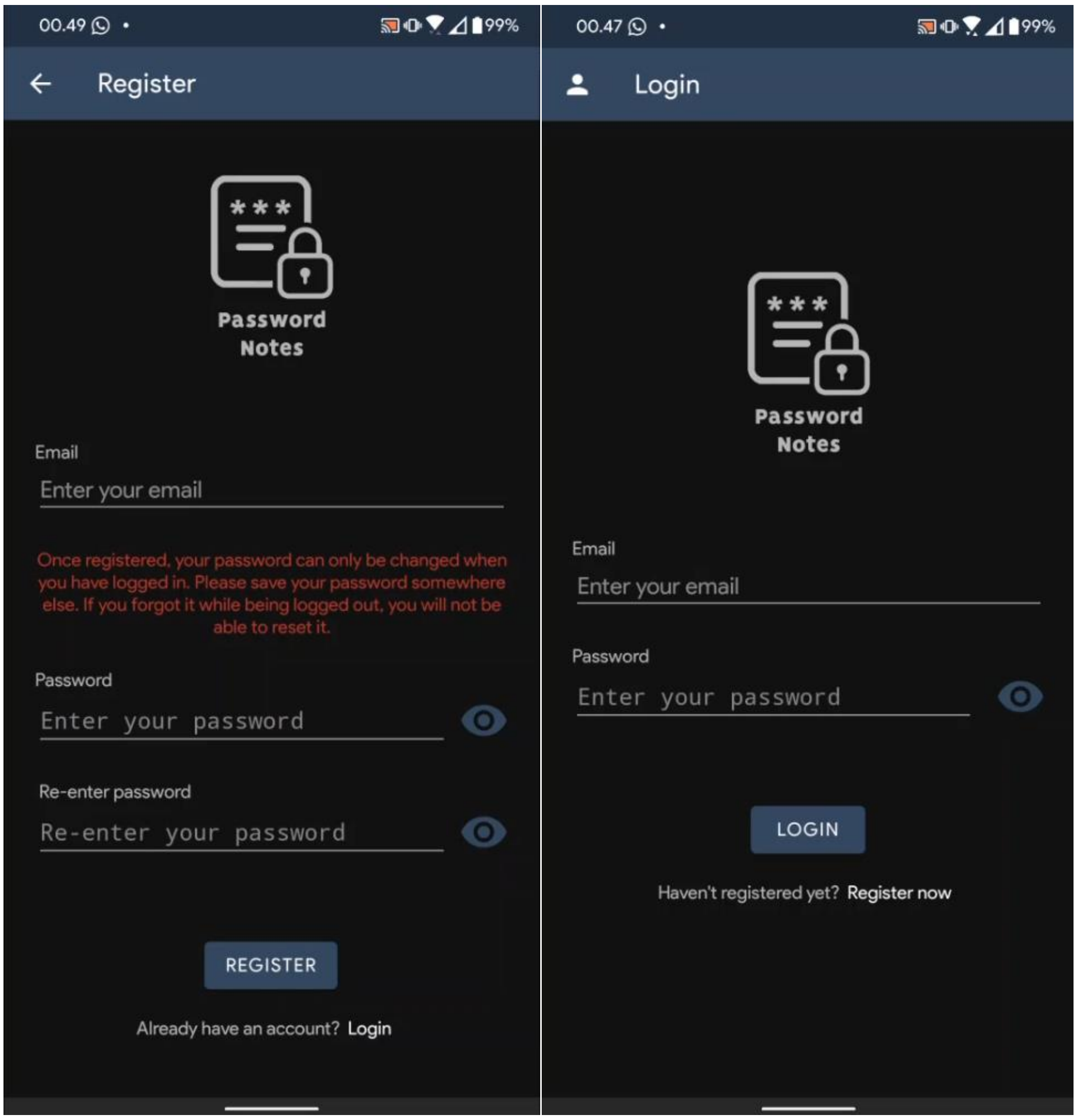
- **Miscellaneous package**

Miscellaneous


com.google.android.material.bottomnavigation.BottomNavigationView.OnNavigationItemSelectedListener
com.google.android.material.bottomnavigation.BottomNavigationView
MainActivity
- navigation : OnNavigationItemSelectedListener
- verify : Runnable
- lock : Runnable
- check_timeout : Runnable
- requireVerify : boolean
- timeouts_value : int[]
- end : long
- timeout : int
- handler : Handler
- bottomnav : BottomNavigationView
- backPressedTime : long
- back : Toast
+ isRequireVerify() : boolean
+ setRequireVerify(requireVerify : boolean) : void
+ getTimeout() : int
+ setTimeout(timeout : int) : void
+ getEnd() : long
+ setEnd(end : long) : void
# onCreate(savedInstanceState : Bundle) : void
- stopRunnable() : void
+ onConfigurationChanged(newConfig : Configuration) : void
+ onPause() : void
+ onResume() : void
+ onBackPressed() : void
androidx.lifecycle.LifecycleObserver
androidx.appcompat.app.AppCompatActivity
java.lang.Runnable
android.widget.Toast
int
int[]
boolean
long
android.app.Application
SecureView
- context : Context
+ onCreate() : void
- setupActivityListener() : void
android.os.Handler
SplashScreen
- logo : ImageView
- time_end : long
- handler : Handler {readOnly}
# onCreate(savedInstanceState : Bundle) : void
android.content.Context
android.widget.ImageView

IV. Application Interface and How the Application Works

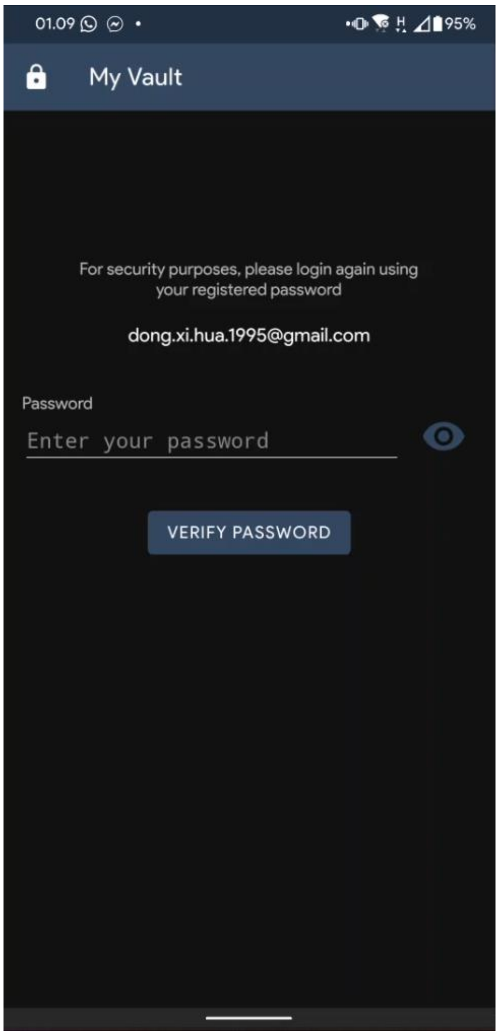
- Register and Login activities



Once registered, your password can only be changed when you have logged in. Please save your password somewhere else. If you forgot it while being logged out, you will not be able to reset it.

These are the *Register* and *Login* activities of the application. When the app is first opened and there is no user logged in, the app will go to the Login activity. When the user taps on the “Register now” text, it will go to the *Register* activity. There, the user can register for a new account. However, if the user is trying to make a new account with an email that is already exists in the database, the app will throw an error displaying to the user that the email entered has already been registered in the database. When the user is registering a new account, they will be prompted to verify their new account via their newly registered email. There will be a link where the user can verify their email in their inbox.

- **Verify Password activity**


01.09 ·95%

My Vault

For security purposes, please login again using your registered password

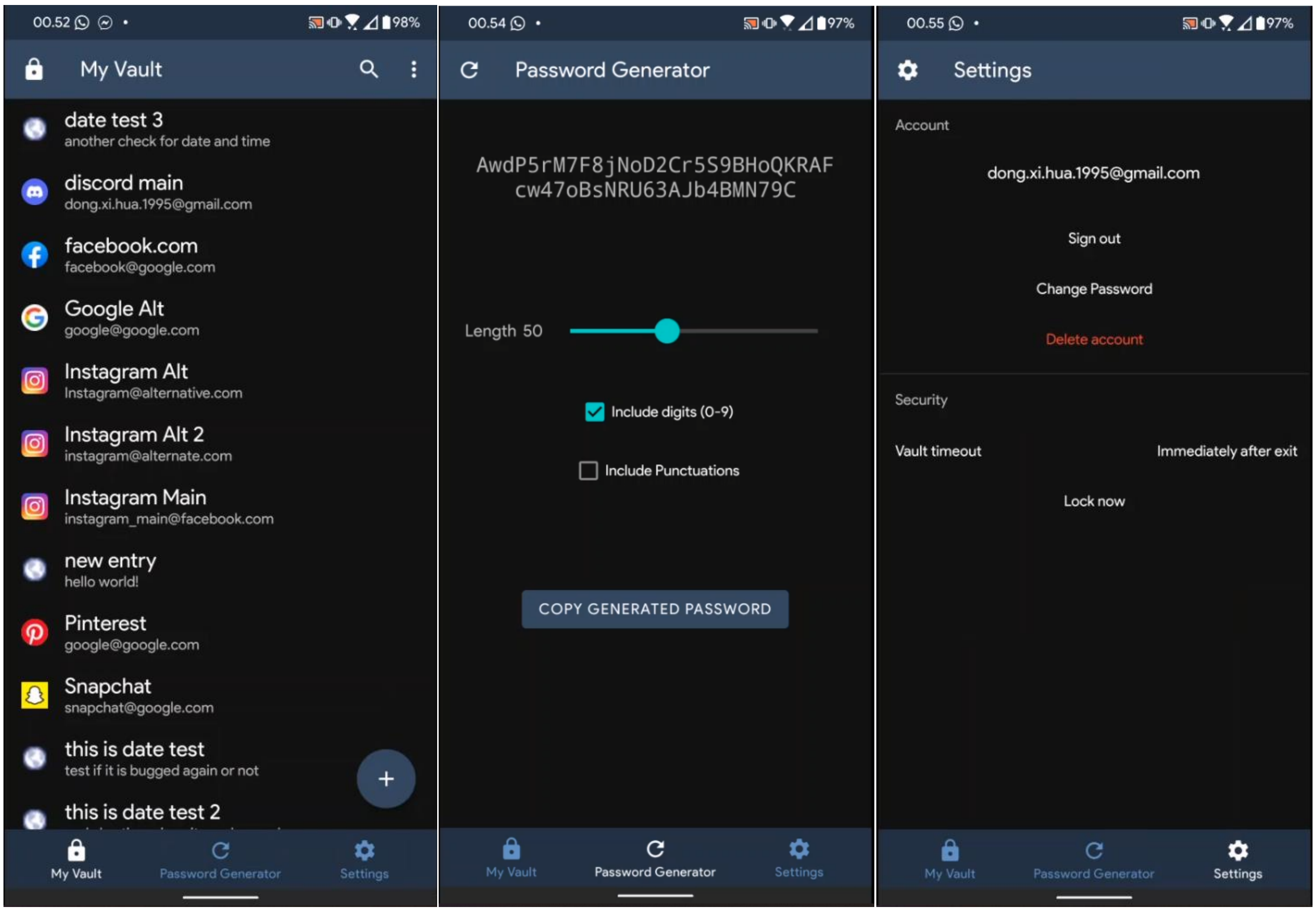
dong.xi.hua.1995@gmail.com

Password

Enter your password

After the user has successfully logged in, they will be redirected to the *VerifyPassword* activity first before they can finally view their passwords. This is an added security so that their password vault will be safe.

- **Password List, Generator, and Account fragments**



date test 3
another check for date and time
discord main
dong.xi.hua.1995@gmail.com
facebook.com
facebook@google.com
Google Alt
google@google.com
Instagram Alt
Instagram@alternative.com
Instagram Alt 2
instagram@alternate.com
Instagram Main
instagram_main@facebook.com
new entry
hello world!
Pinterest
google@google.com
Snapchat
snapchat@google.com
this is date test
test if it is bugged again or not
this is date test 2

AwdP5rM7F8jNoD2Cr5S9BHoQKRAF
cw47oBsNRU63AJb4BMN79C

Length 50

☑ Include digits (0-9)

☐ Include Punctuations

COPY GENERATED PASSWORD

Account

dong.xi.hua.1995@gmail.com

Sign out

Change Password

Delete account

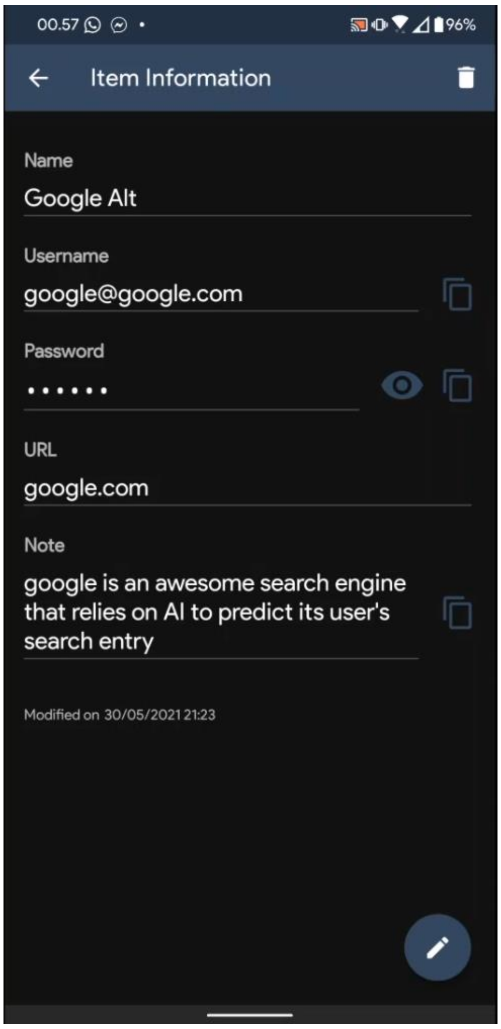
Security

Vault timeout: Immediately after exit

Lock now

After they have entered their password in the *Verify Password* activity, they will be redirected to *PasswordList* activity which is the home fragment of the *Main Activity*. This fragment shows as “My Vault” to the user. There, user can interact with their previously added passwords and or add new ones. This application provides a password generator for the users to ease themselves on an account creation process. The user can access this feature by tapping the “Password Generator” tab on the bottom middle of the app screen. This is the *Generator* fragment in the source code. Lastly, the user can also interact with their account and also the application itself. For account control, there are sign out, change password, and delete account options. For the application control, there are vault timeout and lock now options. Vault timeout is basically a setting to lock the app if the time has exceeded the amount of time set by the user. By default, the app’s timeout is “immediate lock after exit”, which means that if the user exits the app, they will be greeted by the *VerifyPassword* activity when they return to the app.

- **Password View and Password Add/Edit activity**



00.57

96%

Item Information

Name
Google Alt

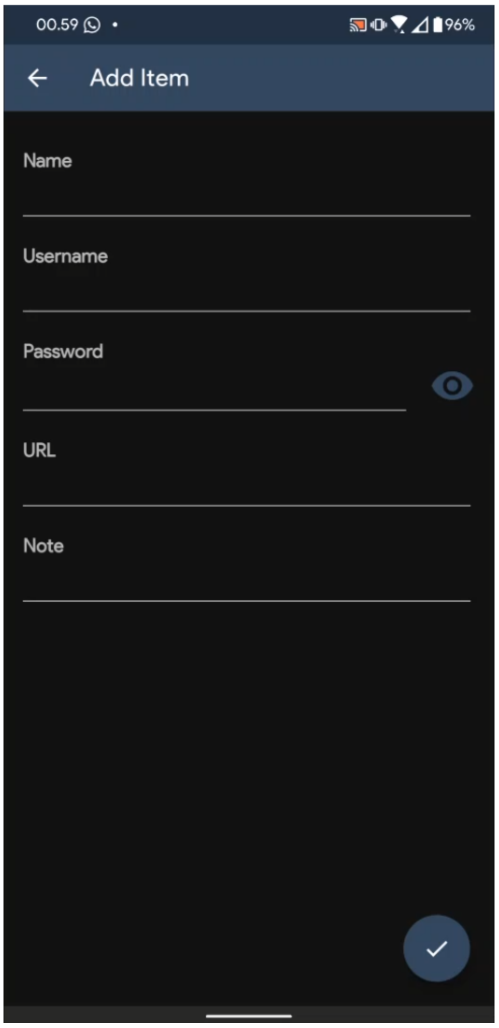
Username
google@google.com

Password
••••••

URL
google.com

Note
google is an awesome search engine that relies on AI to predict its user's search entry

Modified on 30/05/2021 21:23



00.59

96%

Add Item

Name

Username

Password

URL

Note

Moving back to the *PasswordList* fragment, when the user tapped on the plus button on the bottom right of the app, they will be redirected to the "Add Item" activity above. There, the user can add new entry by filling out the details in the text fields that have been provided. When the user tapped on an existing entry, they will be redirected to the "Item Information" activity above, which contains the information about the password entry itself. There, the user can copy the username and password, and also view the password that by default is censored by dots. The user can edit the entry by tapping the pencil icon on the bottom right of the app. There is also an information about when the modification was last made on the entry on the bottom left of the screen, below the “Note” field.

V. Libraries

1. Android SDK 30

   Android SDK 30 is the application development codename for Android 11. This library provides the core elements for Android applications.
2. Firebase

   Firebase is an API used to connect the project to Firebase database. It is by default included in the Android SDK, but it is a standalone library, meaning that it is not part of the core elements for an Android application to be able to run.

VI. Lessons Taken from Doing the Project

From the moment that we were asked to think about what to make for the final project, I instantly thought about an Android application. I do not know why but I was so intrigued on Android development. I wanted to make an Android application for the sake of my curiosity on the world of the Android development. And so, I found out that this is the greatest opportunity for me to dive into the world of Android application development. Without hesitation, I wrote in the Google Docs proposal that I wanted to make an Android application, whatever the application will be. I began to look for inspiration, and a password manager application seemed to match my interest. And so, without a single knowledge of Android Studio, I began coding by looking tutorials online. There were ideas that were scrapped because there was not enough time: the ability to login by using fingerprint and the ability to autofill login and password fields on apps and websites (I have looked everywhere, but there were no one that discussed this topic). There are still other projects that needs to be done. Android Studio was totally a new thing for me. I did not know anything related to it. And so, this was the biggest difficulty that I faced. Other than that, the confusing “deprecated” signs that appears on some methods made me confused on what alternatives to use, because the methods being marked as “deprecated” are the ones that are used in the tutorials. The lessons that I can take from this project is that Android development sure is fun, but sometimes it can be a tedious work because something that looks so simple is actually so hard to implement behind the screen. One that might think is simple but very hard to implement is the dropdown selectable items list, like in the device settings app. That one was very tedious to make. From this fact, I can appreciate Android developers even more who made majestic app UIs and animations.

VII. Project Technical Description

Password Notes is an application that can take note of user inputted credentials that are used in other services. The credentials that have been submitted as entries are going to be stored in a database. The database used for storing the entries is Firebase, a database provider developed by Google (the database structure is different from MySQL. It is simpler and more friendly to be used especially for newbies). Password Manager is an Android application developed in Android Studio using Android SDK 30. In other words, it was developed using Android 11's SDK. The features of Password Manager are:

1. Make new entries.

   The said entry is in a form of user credentials used to log in to other services other than Password Manager. Duplicate entries are allowed since this application is not a specialized application such as dictionary. New entries will be automatically sorted in descending order alphabetically.
2. Edit and delete entries.

   Entries that previously have been added by user can be deleted or edited, depending on the user's need.
3. Search entries

   When the entries are growing in size, it can be a tedious work to find a specific entry in the list. To ease the user on finding the entry that they need, Password Notes has successfully implemented a search function to filter the entry that the user is currently looking for from the rest of the entries.
4. Generate password.

   Password Notes can generate a new password with custom options depending on user's needs. User has three options for the password to be generated:

   a. Include digits.

      User has the freedom to include digits in the generated password. If this option is not checked, the generated password will not contain any digits.

   b. Include punctuations.

      User also has the freedom to include or exclude punctuations in the generated password. If this option is not checked, the generated password will not contain punctuations.

   c. Length

      Finally, user has the freedom to set the length of the generated password. The minimum length that the user can set is 6 characters, and the maximum length of the generated password is 128 characters.

5. Full account control.

   To use this application, user needs to register for a new account or log in into an existing one. After registering, user has the freedom to sign out from the application, change password, and delete their account. Due to security reasons, user can only change their password once they have logged in to their account. If user forget their password while being logged out from their account, they will lose all their entries. It is highly advised that user always remember their password or store it somewhere safe.

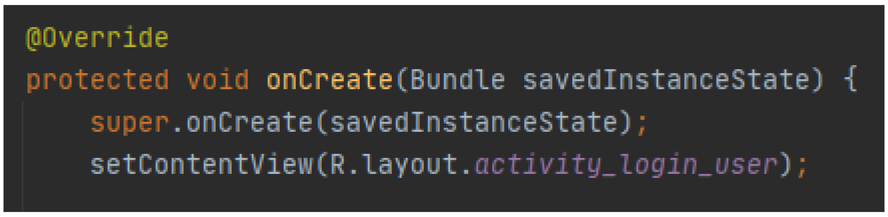
6. Application control

   The user has control over when their password vault will be locked. The timeout that the user can choose are: 1 minute, 5 minutes, 10 minutes, 15 minutes, 20 minutes, 30 minutes, and immediately after application exits. The last option, which is the immediate lock after application exit, means that whenever user leaves the application, even if they only let the application goes into background process, they will be prompted to enter their password again once they open Password Notes again. The other options will do the same, however, they will only be prompted to enter their password again if the time has exceeded the time option that the user has chosen. Password Manager chooses immediate lock after exit as the default option.

Password Notes will not run if the target operating system is below Android KitKat (Version 4.4; Android SDK 19). As for the reason why version 4.4 was chosen as the minimum, the parent class' methods and interfaces are much different from the previous versions. As of today, Android SDK 19 is considered as the minimum SDK for major app releases to run, as devices running previous versions of Android before version 4.4 are decreasing and is considered to be deprecated.

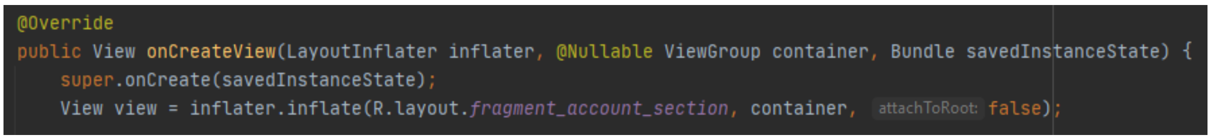
VIII. Codes and Features Explanation

1. Base codes



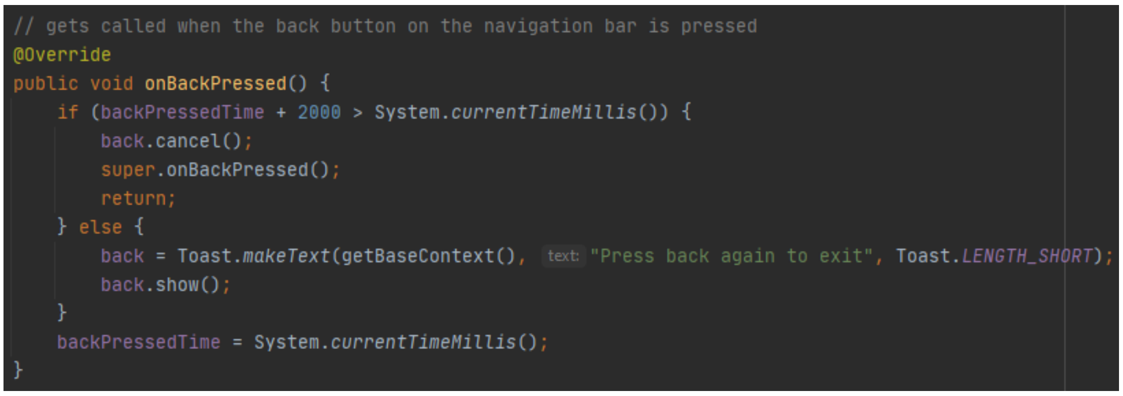
```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_login_user);
```

onCreate() the most important override method that will always be called whenever a new activity opens. It is similar to a constructor of a class, but this is a "constructor" for each activity. Every activity present in an application will always have this *onCreate()* override method in the beginning of the class. If this method is not called, the application surely will crash upon starting the activity.



```java
@Override
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    View view = inflater.inflate(R.layout.fragment_account_section, container,  attachToRoot: false);
```

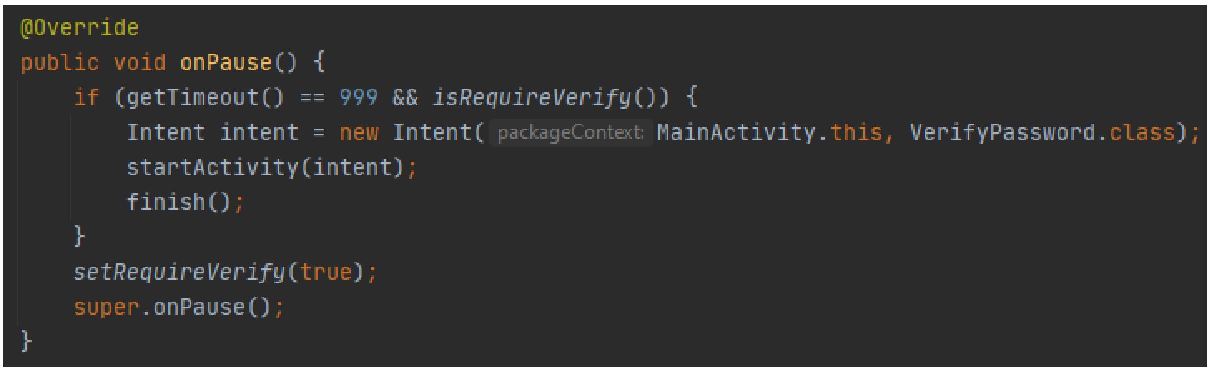
In Android, there is "another type" of activity called fragment. Its behavior differs from a normal activity. Think of fragment as a part of an activity, which means that fragments are activities inside an activity. Fragment can be used to achieve convenience in application usage, such as user does not need to press on buttons to do something, instead they only need to press on icons in a navigation bar to go to an activity. One thing that differs from a normal activity is that the *onCreate()* options is 'replaced' by this *onCreateView()* method. Although the parent class is the same for both methods (*onCreate()* and *onCreateView()*), if the programmer is to put the



```java
// gets called when the back button on the navigation bar is pressed
@Override
public void onBackPressed() {
    if (backPressedTime + 2000 > System.currentTimeMillis()) {
        back.cancel();
        super.onBackPressed();
        return;
    } else {
        back = Toast.makeText(getBaseContext(),  text: "Press back again to exit", Toast.LENGTH_SHORT);
        back.show();
    }
    backPressedTime = System.currentTimeMillis();
}
```

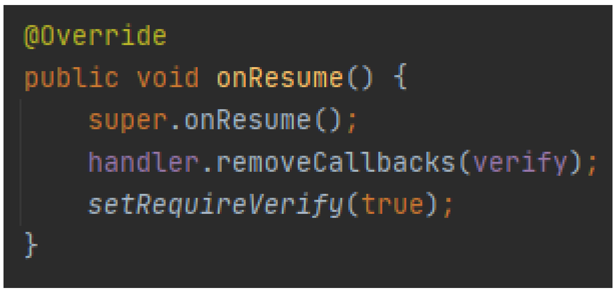
‘constructor’ for a fragment in the *onCreate()* method, the application will crash.

onBackPressed() is a method to override the behavior of the activity whenever the back button is pressed. This method is derived from the parent class *AppCompatActivity*, which is the base class for the activity to run. In the code snippet above, it will show a message saying “Press back again to exit” when the back button is pressed. This method is only overridden in certain activities to maximize user’s convenience.



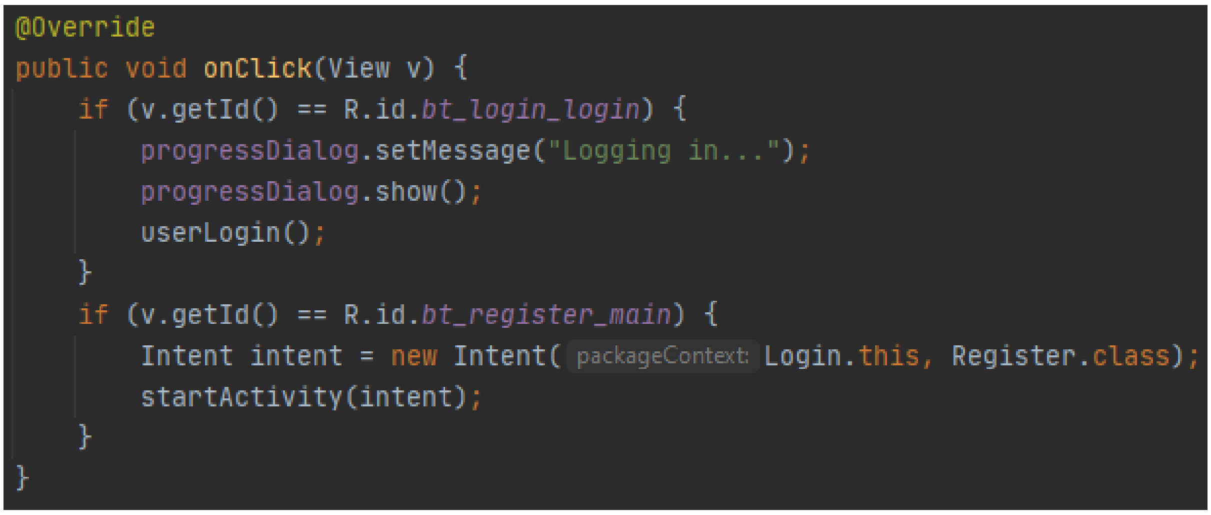
```
@Override
public void onPause() {
    if (getTimeout() == 999 && isRequireVerify()) {
        Intent intent = new Intent( packageContext: MainActivity.this, VerifyPassword.class);
        startActivity(intent);
        finish();
    }
    setRequireVerify(true);
    super.onPause();
}
```

onPause() is called when an activity goes on paused state. For example, if we open the settings application in our android phone, it will open a new activity which contains all of the options that we can see and choose. When we clicked on one of the options, a new activity will open, and the previous activity will go on a paused state. This method is optional to be overridden, but in some activities in Password Notes, *onPause()* needs to be overridden to change some behavior of the application. For example, in the code snippet above, the *onPause()* method overrides the default behavior of the activity when it goes on pause state, which will do nothing. But because it has been overridden, whenever the activity that overrides the *onPause()* method goes on pause, it will do the things that have been defined in the method. In Password Note’s case, whenever the user leaves the application, when they open the application again, they will be prompted to enter their password again to access the vault. This behavior can change again depending on the user’s option on when to lock their password vault.



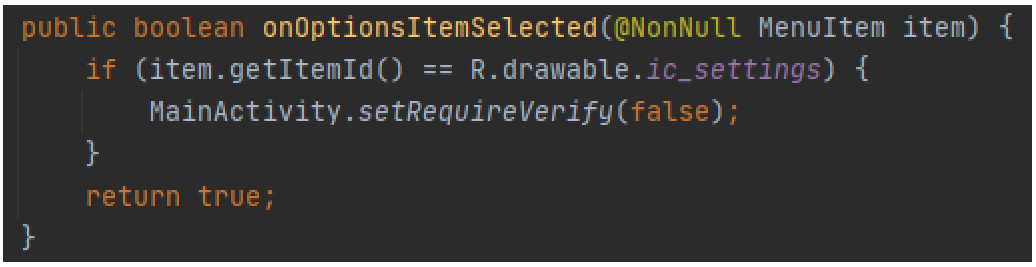
```java
@Override
public void onResume() {
    super.onResume();
    handler.removeCallbacks(verify);
    setRequireVerify(true);
}
```

Corresponding to the *onPause()* method, *onResume()* is a method that is called when the previous activity has resumed. Continuing from the previous example, after the user has finished doing things in the current activity and presses the back button, the main activity that contains all of the options for the device settings will be resumed. When this happens, *onResume()* will be called. *onResume()* is also optional to be overridden as it is not a mandatory method to be overridden such as *onCreate()*. The default behavior of *onResume()* is the same as *onPause()*, which will do nothing if it is not overridden.



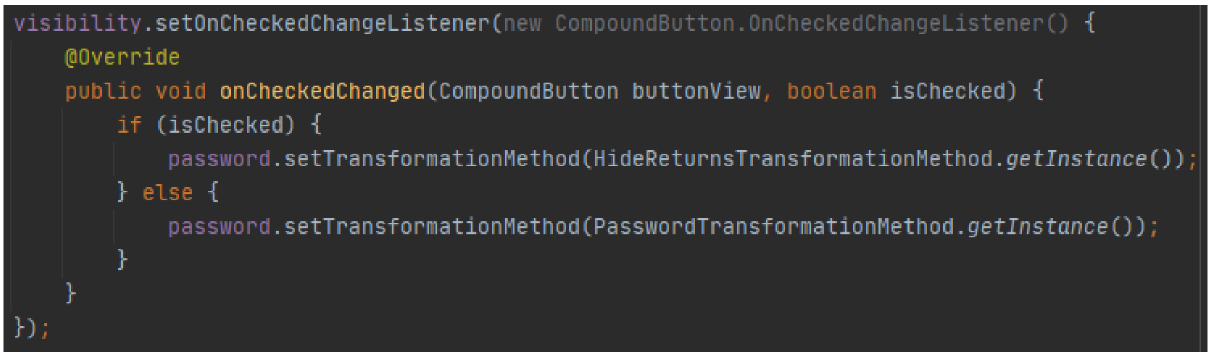
```java
@Override
public void onClick(View v) {
    if (v.getId() == R.id.bt_login_login) {
        progressDialog.setMessage("Logging in...");
        progressDialog.show();
        userLogin();
    }
    if (v.getId() == R.id.bt_register_main) {
        Intent intent = new Intent( packageContext: Login.this, Register.class);
        startActivity(intent);
    }
}
```

onClick() is an interface that will be called whenever something is meant to be pressed. Activities that contain pressable things will be guaranteed to have this interface implemented.



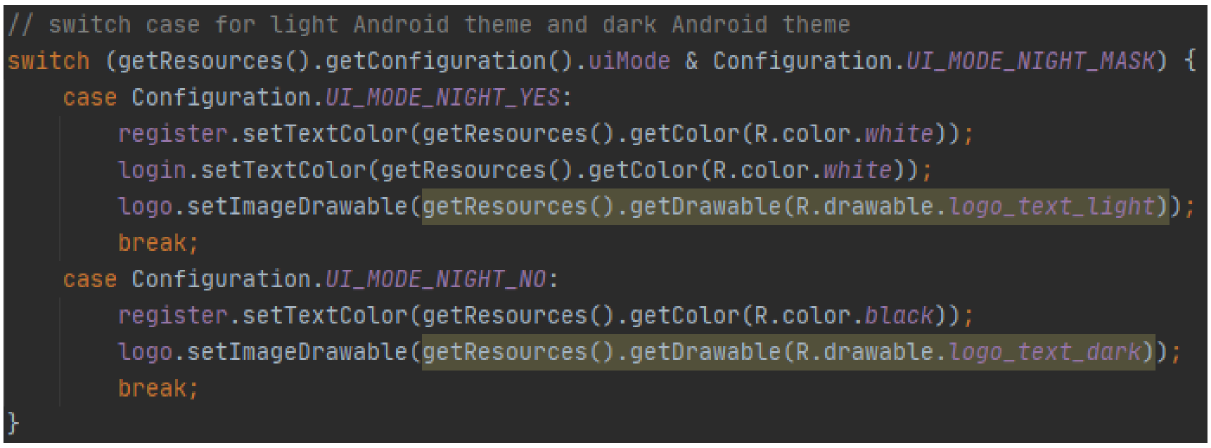
```java
public boolean onOptionsItemSelected(@NonNull MenuItem item) {
    if (item.getItemId() == R.drawable.ic_settings) {
        MainActivity.setRequireVerify(false);
    }
    return true;
}
```

onOptionsItemSelected() is a method that will be called whenever the user chooses a menu option that is available in the menu settings in the action bar.



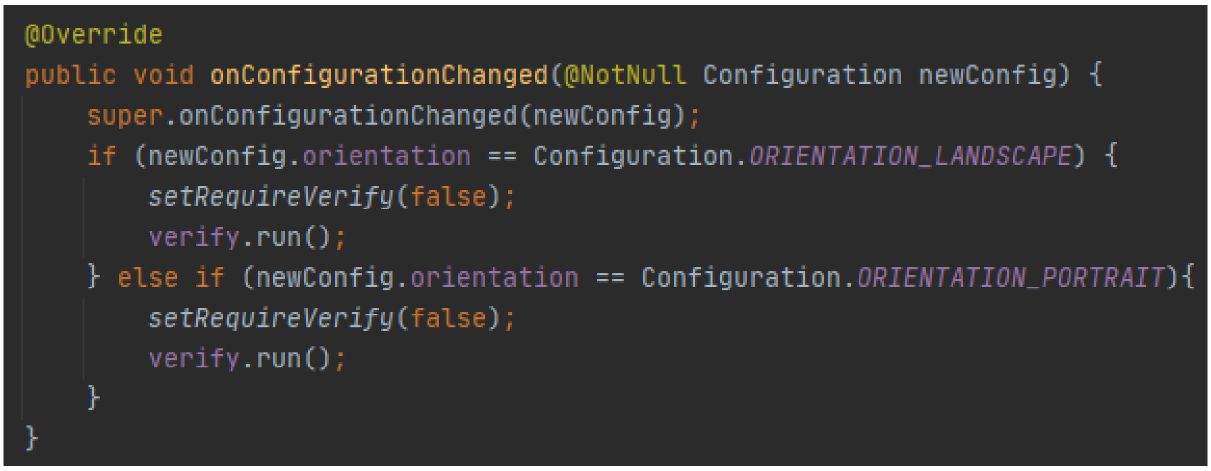
```
visibility.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        if (isChecked) {
            password.setTransformationMethod(HideReturnsTransformationMethod.getInstance());
        } else {
            password.setTransformationMethod(PasswordTransformationMethod.getInstance());
        }
    }
});
```

onCheckedChanged() is an interface that is called when a toggle button is pressed. If the value is 1 or true, it will go to the *isChecked() if* statement, else it will go to the ‘else’ statement. In the code snippet above, the toggle button is used as a toggle to view or hide the password fields in activities. If the button is checked, or toggled on, it will show what the user has entered into the field. If it is toggled of or not checked, then it will go to its default state, that is hiding the password from the user view. The password will be shown as dots to the user.



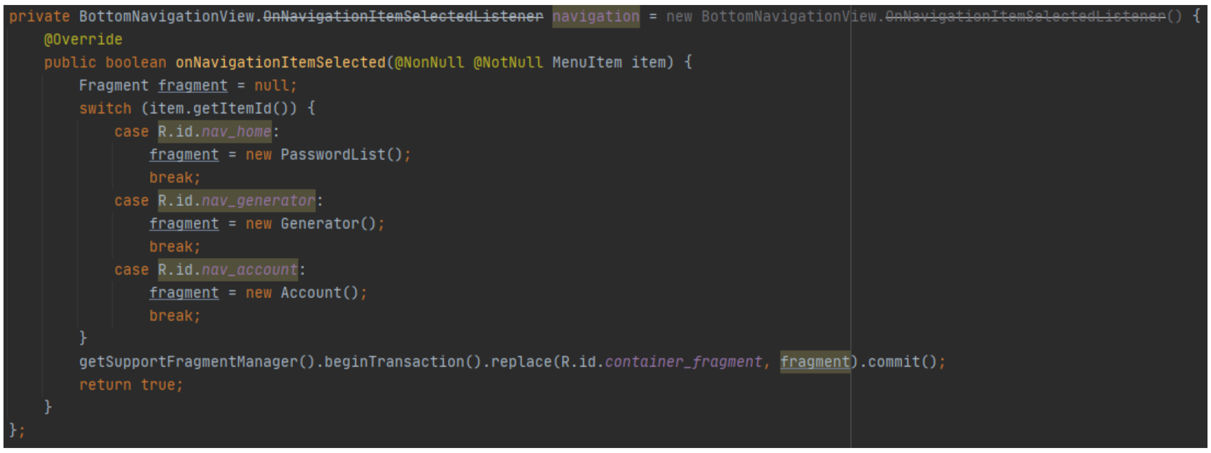
```
// switch case for light Android theme and dark Android theme
switch (getResources().getConfiguration().uiMode & Configuration.UI_MODE_NIGHT_MASK) {
    case Configuration.UI_MODE_NIGHT_YES:
        register.setTextColor(getResources().getColor(R.color.white));
        login.setTextColor(getResources().getColor(R.color.white));
        logo.setImageDrawable(getResources().getDrawable(R.drawable.logo_text_light));
        break;
    case Configuration.UI_MODE_NIGHT_NO:
        register.setTextColor(getResources().getColor(R.color.black));
        logo.setImageDrawable(getResources().getDrawable(R.drawable.logo_text_dark));
        break;
}
```

The switch statement in the code snippet above is declared to let the application know whether the phone is currently in dark theme or light theme. If it is currently in dark theme, the switch statement will go into the *UI_MODE_NIGHT_YES* case. If it is in light theme, then it will go to the *UI_MODE_NIGHT_NO* case.



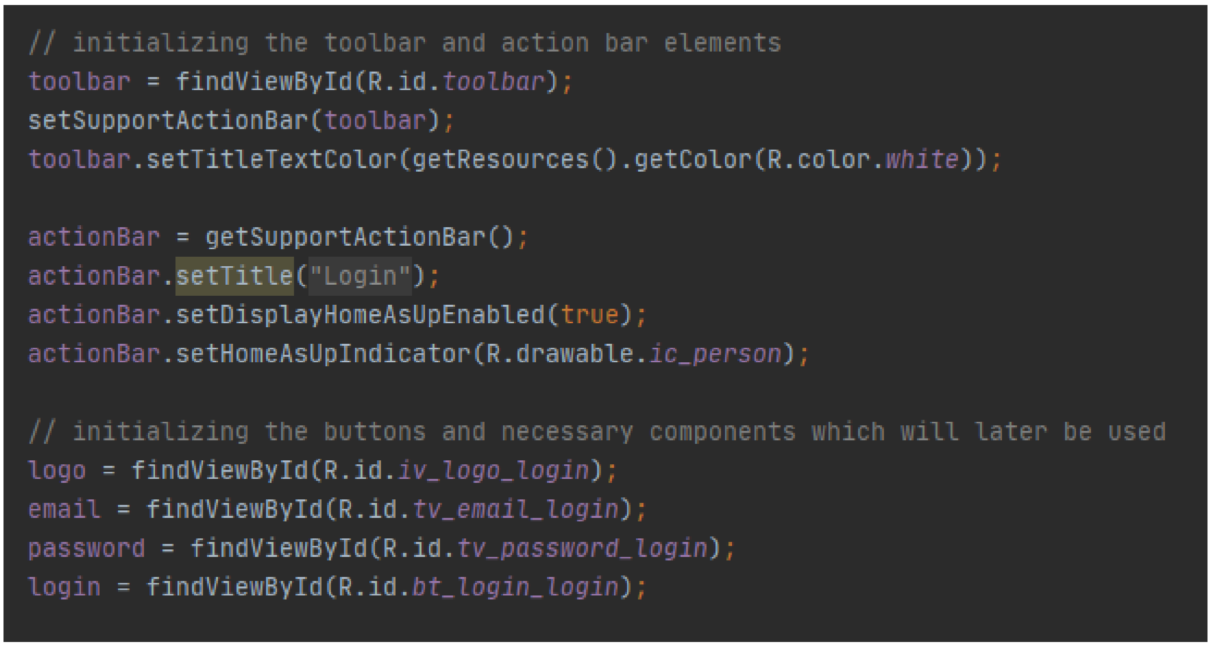
```java
@Override
public void onConfigurationChanged(@NotNull Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
        setRequireVerify(false);
        verify.run();
    } else if (newConfig.orientation == Configuration.ORIENTATION_PORTRAIT){
        setRequireVerify(false);
        verify.run();
    }
}
```

onConfigurationChanged() is a method to check whether the configuration of the device is changed. The configuration here means the orientation of the device. Whenever the orientation of the device changes, the current activity that is being displayed to the user will go on a paused state, thus calling the *onPause()* method. After the orientation have changed, the *onResume()* method will then be called. This process may not be realized by user as it is executed in just a matter of split seconds. Other than when the orientation changes, the whole process, along with *onConfigurationChanged()* will also be executed when the application goes on a split screen mode. This mode is present since the release of Android Nougat (version 7.0; Android SDK 24). *onConfigurationChanged()* is an optional method to override. It may or may not be overridden depending on the application needs.



```java
private BottomNavigationView.~~OnNavigationItemSelectedListener~~ navigation = new BottomNavigationView.~~OnNavigationItemSelectedListener~~() {
    @Override
    public boolean onNavigationItemSelected(@NonNull @NotNull MenuItem item) {
        Fragment fragment = null;
        switch (item.getItemId()) {
            case R.id.nav_home:
                fragment = new PasswordList();
                break;
            case R.id.nav_generator:
                fragment = new Generator();
                break;
            case R.id.nav_account:
                fragment = new Account();
                break;
        }
        getSupportFragmentManager().beginTransaction().replace(R.id.container_fragment, fragment).commit();
        return true;
    }
};
```

onNavigationItemsSelected() is an interface specialized for bottom navigation that is present in Password Notes. It will handle the item (icon) presses whenever the user clicks on them.



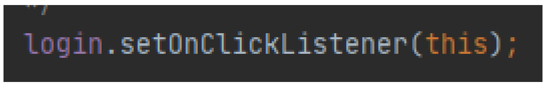
```
// initializing the toolbar and action bar elements
toolbar = findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
toolbar.setTitleTextColor(getResources().getColor(R.color.white));

actionBar = getSupportActionBar();
actionBar.setTitle("Login");
actionBar.setDisplayHomeAsUpEnabled(true);
actionBar.setHomeAsUpIndicator(R.drawable.ic_person);

// initializing the buttons and necessary components which will later be used
logo = findViewById(R.id.iv_logo_login);
email = findViewById(R.id.tv_email_login);
password = findViewById(R.id.tv_password_login);
login = findViewById(R.id.bt_login_login);
```

To ease the explanation, the code snippet above will be divided into three parts according to the respective empty lines. I decided to put one example of the above code in the base code section because the code pattern above is used in the *onCreate()* method in all activities and fragments in this application.

The first one is the toolbar. An Android application, by default, already has a toolbar on its own, but it is considered to be deprecated and is not recommended by Google themselves as it is hard to modify them. Instead, they recommended programmers to implement a custom toolbar which can be customized freely according to the needs of the application. Hence, therefore I always declare a new toolbar variable in every single activity and fragment. Once the toolbar has been set, the next thing to set is the action bar.

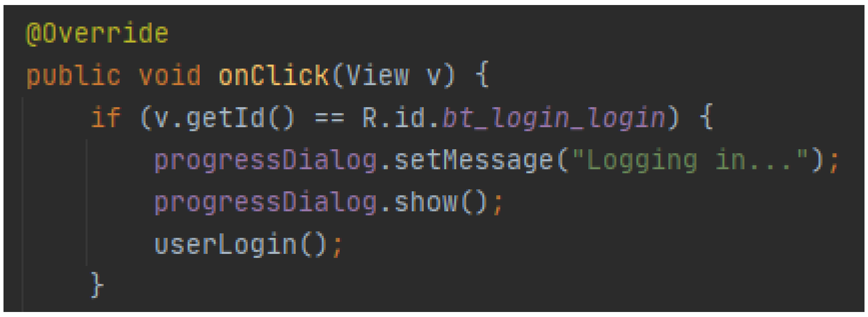
Moving on to the next part of the code snippet, the action bar is there as a support and holder at the same time for items that are going to be added into the toolbar. For example, there is a search button in the toolbar to search the entries in the Vault fragment of the Password Notes application. The action bar goes into play to make the search button appear in the toolbar.

The final part of the code snippet is the items initialization. For the items in the activity, such as buttons and text fields to be able to function, the programmer needs to let the compiler know what function will be executed when a button is pressed, or what text to set in the text field when the activity is launched for the first time. An example of this is the login button. In the code snippet above, the login button has been set to connect to the *bt_login_login* resource. Somewhere in the code, there will be a line that says this:



```
login.setOnClickListener(this);
```

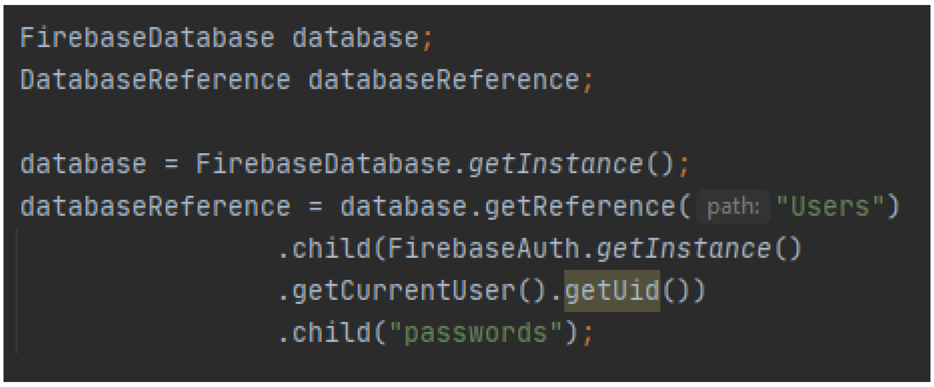
This tells the compiler that when the button is clicked, it must do a certain function. That function of the login button is:



```
@Override
public void onClick(View v) {
    if (v.getId() == R.id.bt_login_login) {
        progressDialog.setMessage("Logging in...");
        progressDialog.show();
        userLogin();
    }
```

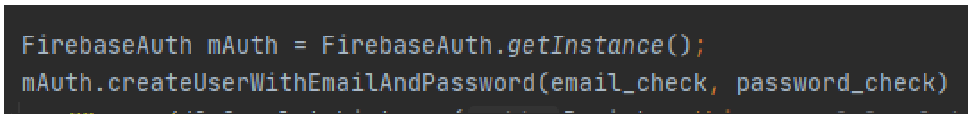
which will log the user in by calling the *userLogin()* method. This method will later be explained in its respective section of this report.

2. **Firebase**



```
FirebaseDatabase database;
DatabaseReference databaseReference;

database = FirebaseDatabase.getInstance();
databaseReference = database.getReference( path: "Users")
                .child(FirebaseAuth.getInstance()
                .getCurrentUser().getUid())
                .child("passwords");
```

The first two classes which are part of the Firebase package that are used in this application are *FirebaseDatabse* class and *DatabaseReference* class.



```
FirebaseAuth mAuth = FirebaseAuth.getInstance();
mAuth.createUserWithEmailAndPassword(email_check, password_check)
```

FirebaseDatabase is there to get the instance of the Firebase database that has been connected to the project. The *DatabaseReference* is where the database path will be stored. In the code snippet above, the database reference is referencing to the path of root > Users > current account > passwords.

The final class which is a part of the Firebase package is the *FirebaseAuth* class. As the name suggests, this class provide the ability for the application to log in and or register a new account by using a pair of email and password. The validation of the email and password pattern will have to be done by the programmer as the class only provide the ability to register a new user and check the credentials. If the credentials are incorrect, it will simply throw an exception which can later be caught to show an error message to the user.

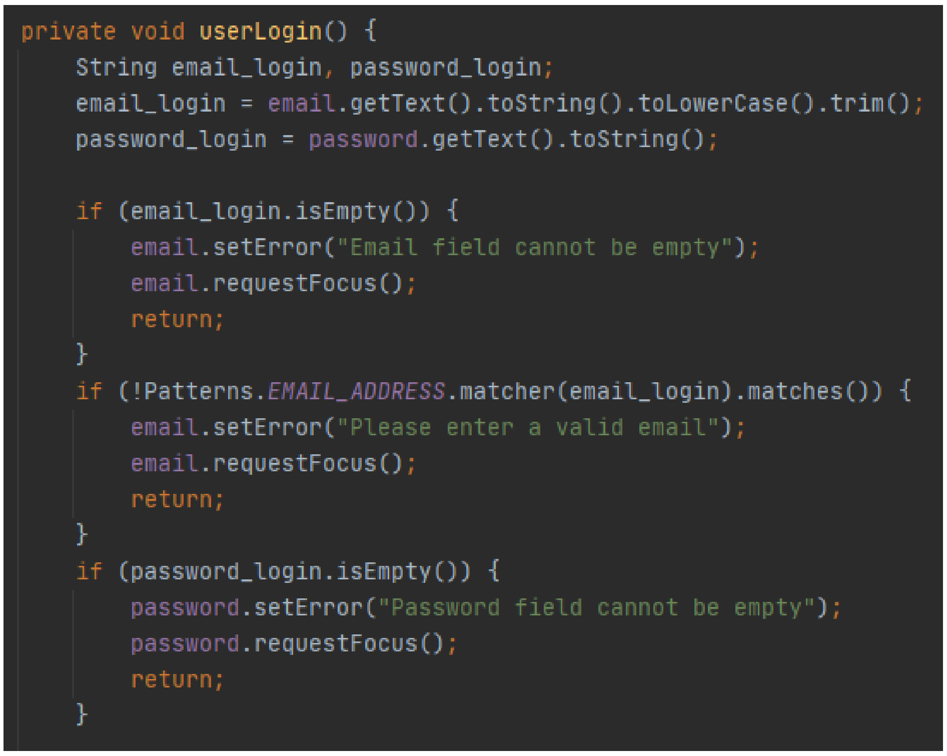
3. Packages

- Activity and Fragment packages

The activity and fragment packages are the packages of user activities. For example, in the activity package, there is a class called *Login*. That class contains the activity that will display the login screen to the user. In that screen, user can input their email and password to login into the app. As the name suggests, Fragment means that the classes contained in that package are fragments of the Main Activity of the application. There are three fragments in total, one is *PasswordList* which will display the inputted passwords by the user, the other one is *Generator*, that includes a password generator for the user if they are struggling to make a new password. Lastly, *Account* is where the user can interact with their account.

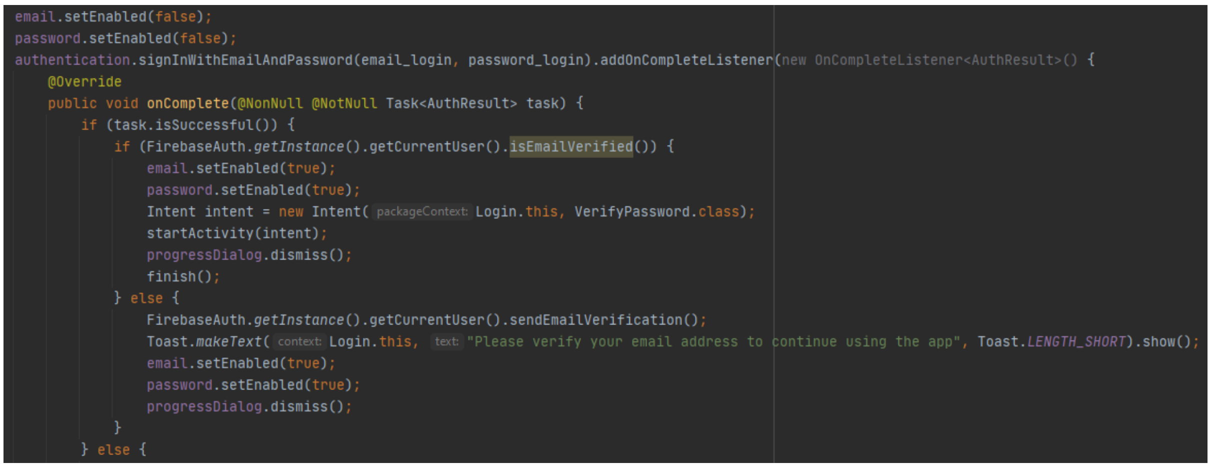
a. Login Class

Continuing from the example given in the base code section of this report, there is a method called userLogin(), which will log the user in if the credentials entered by the user matches the credentials stored in the database.



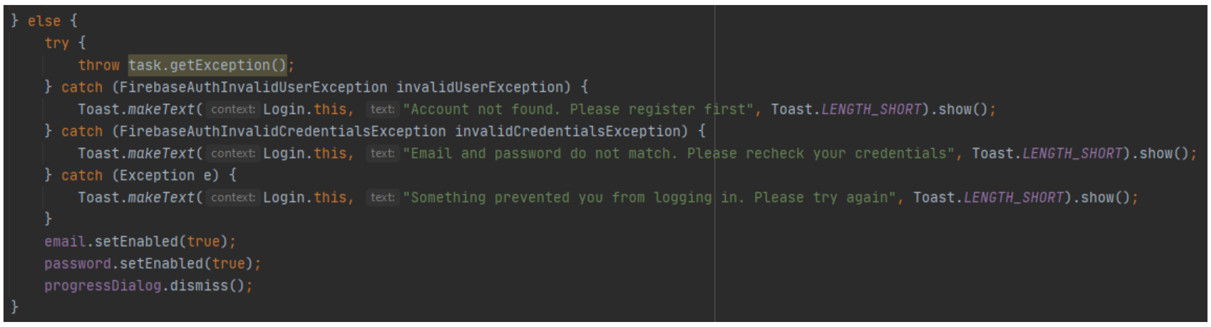
```java
private void userLogin() {
    String email_login, password_login;
    email_login = email.getText().toString().toLowerCase().trim();
    password_login = password.getText().toString();

    if (email_login.isEmpty()) {
        email.setError("Email field cannot be empty");
        email.requestFocus();
        return;
    }
    if (!Patterns.EMAIL_ADDRESS.matcher(email_login).matches()) {
        email.setError("Please enter a valid email");
        email.requestFocus();
        return;
    }
    if (password_login.isEmpty()) {
        password.setError("Password field cannot be empty");
        password.requestFocus();
        return;
    }
```

The first part of the *userLogin()* method instruction is checking the input by the user. It will first check the fields. If the fields are empty, simply stop the process and throw an error to the user stating that the fields must not be empty. The second instruction is to check whether the email entered by the user matches the email pattern. To ease the process, I used the Patterns class provided by the android.util package. If all of the conditions are passed, then the program will go to the second part of the instruction.



```java
email.setEnabled(false);
password.setEnabled(false);
authentication.signInWithEmailAndPassword(email_login, password_login).addOnCompleteListener(new OnCompleteListener<AuthResult>() {
    @Override
    public void onComplete(@NonNull @NotNull Task<AuthResult> task) {
        if (task.isSuccessful()) {
            if (FirebaseAuth.getInstance().getCurrentUser().isEmailVerified()) {
                email.setEnabled(true);
                password.setEnabled(true);
                Intent intent = new Intent( packageContext: Login.this, VerifyPassword.class);
                startActivity(intent);
                progressDialog.dismiss();
                finish();
            } else {
                FirebaseAuth.getInstance().getCurrentUser().sendEmailVerification();
                Toast.makeText( context: Login.this,  text: "Please verify your email address to continue using the app", Toast.LENGTH_SHORT).show();
                email.setEnabled(true);
                password.setEnabled(true);
                progressDialog.dismiss();
            }
        } else {
```

The second instruction is to log the user in with the credentials that have been provided by the user in the text fields. The *email.setEnabled()* and *password.setEnabled()* are called to disable the email and password input text field. This is simply to dismiss the keyboard so that it is not blocking the log in process. After that, the method will call the *FirebaseAuth.signInWithEmailAndPassword()* (*authentication* is the *FirebaseAuth* object in the code snippet) method that takes the email and password String as parameters. It will send a request to the Firebase database that have been connected to the project to search for a matching login credentials. While requesting this, a loading dialog will show to let the user know that the application is currently doing something. Keep in mind that the user's email must be verified first in order for them to log into the app. If the email is not yet verified, they will get an error saying that they need to verify their email first to continue the log in process. If they have verified their email and the process is successful, it will log the user in by calling the *Verify Password* activity in the *Intent*

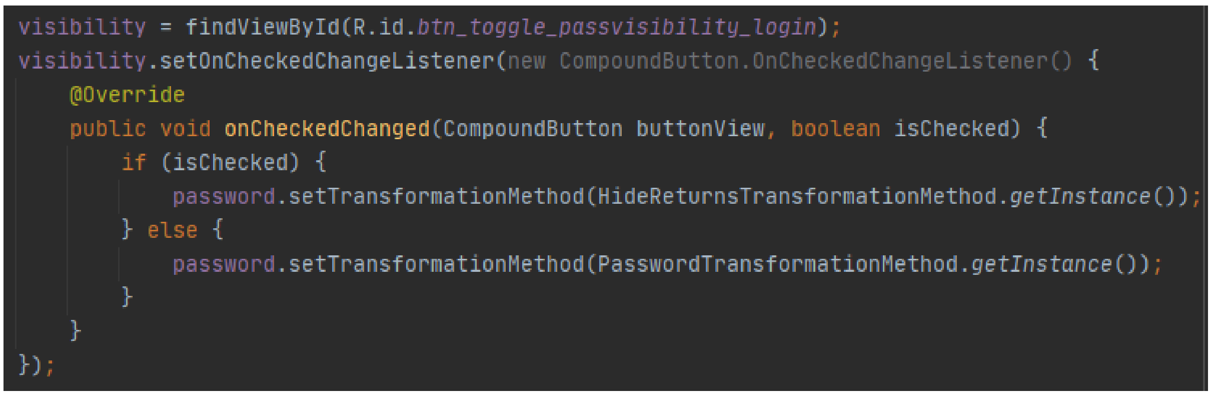


```
} else {
    try {
        throw task.getException();
    } catch (FirebaseAuthInvalidUserException invalidUserException) {
        Toast.makeText( context: Login.this,  text: "Account not found. Please register first", Toast.LENGTH_SHORT).show();
    } catch (FirebaseAuthInvalidCredentialsException invalidCredentialsException) {
        Toast.makeText( context: Login.this,  text: "Email and password do not match. Please recheck your credentials", Toast.LENGTH_SHORT).show();
    } catch (Exception e) {
        Toast.makeText( context: Login.this,  text: "Something prevented you from logging in. Please try again", Toast.LENGTH_SHORT).show();
    }
    email.setEnabled(true);
    password.setEnabled(true);
    progressDialog.dismiss();
}
```

intent = new Intent() line. However, if the process failed, it will go to the *else* statement below.

The method then tries to get the exception thrown by the authentication process. If it catches the *FirebaseAuthInvalidUserException* error, it will show an error to the user saying that the email entered by the user is not registered yet on the database. If it catches the *FirebaseAuthInvalidCredentialsException* error, it will show an error to the user saying that the credentials entered by them previously did not match any data that are stored in the database. If it catches neither of the errors, it will show an error that is not related to both of them. Other errors include the Firebase servers are down, user’s connection is insufficient, and so on. After it catches the exceptions, it will then re-enable the text fields that were previously disabled. Both the successful and the failure statements will dismiss the loading dialog afterwards.

The password text field has a toggle button to show and hide the password currently being entered by the user. Below is the code that makes it possible to do so:

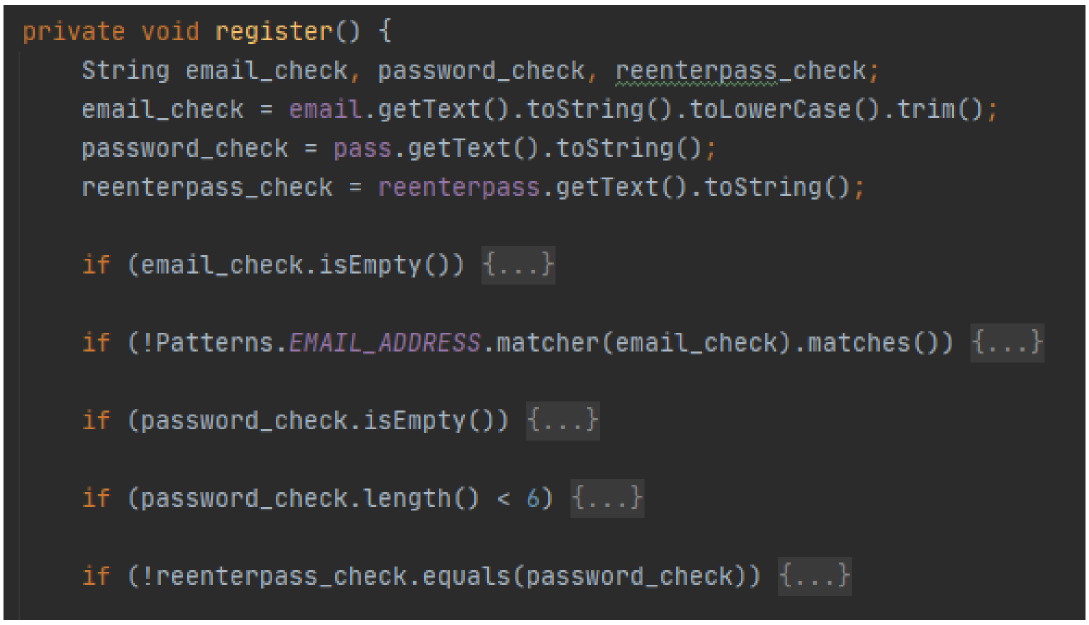


```
visibility = findViewById(R.id.btn_toggle_passvisibility_login);
visibility.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        if (isChecked) {
            password.setTransformationMethod(HideReturnsTransformationMethod.getInstance());
        } else {
            password.setTransformationMethod(PasswordTransformationMethod.getInstance());
        }
    }
});
```

When the toggle button is activated, it will show the unhidden password. When the toggle button is deactivated, it will show the hidden password, which is the usual dots that we see when we are entering sensitive information such as password and pin.

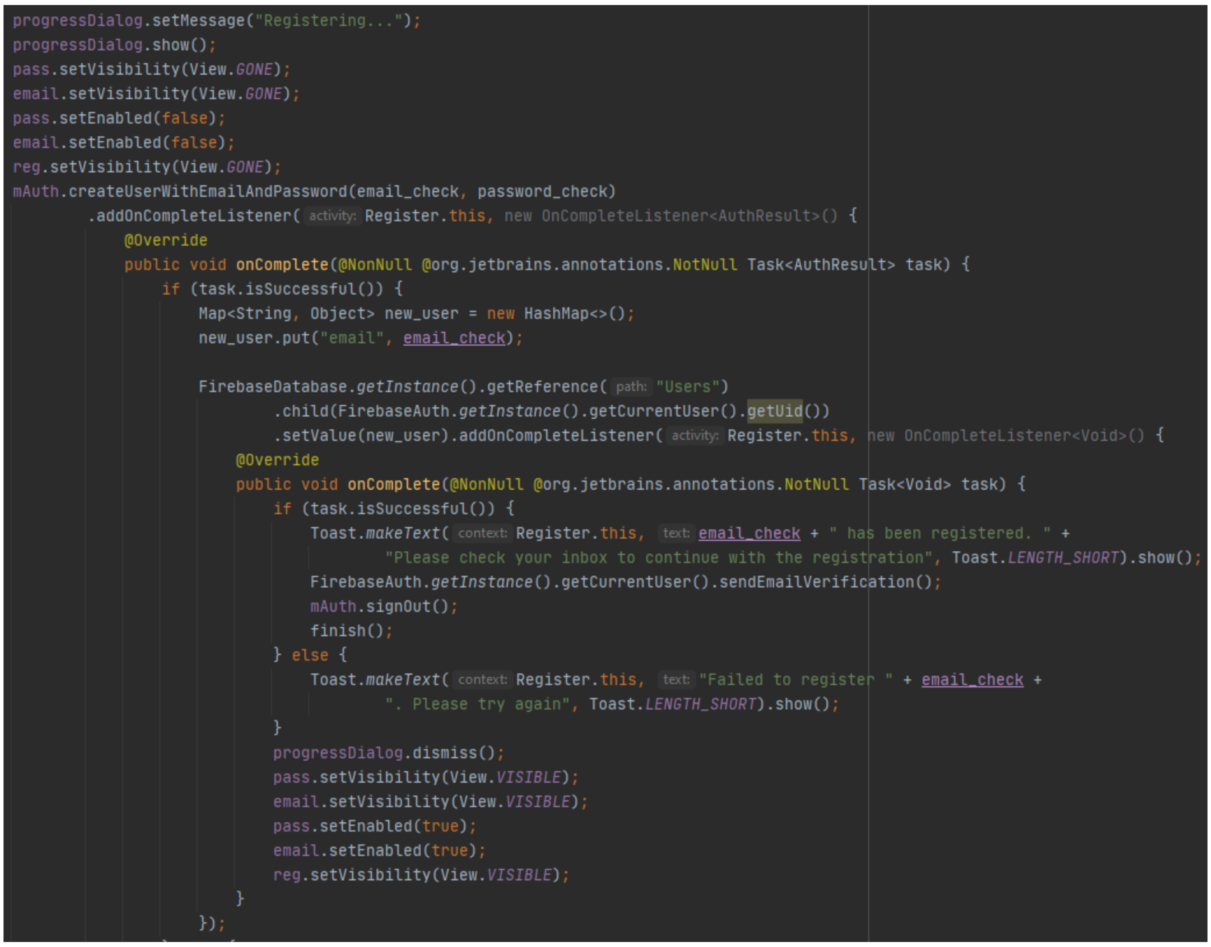
b. Register Class

The register class is basically the same as the Login class, but this class is there as a way to register a new user. The differences between this class and the Login class are shown below.



```
private void register() {
    String email_check, password_check, reenterpass_check;
    email_check = email.getText().toString().toLowerCase().trim();
    password_check = pass.getText().toString();
    reenterpass_check = reenterpass.getText().toString();

    if (email_check.isEmpty()) {...}

    if (!Patterns.EMAIL_ADDRESS.matcher(email_check).matches()) {...}

    if (password_check.isEmpty()) {...}

    if (password_check.length() < 6) {...}

    if (!reenterpass_check.equals(password_check)) {...}
```

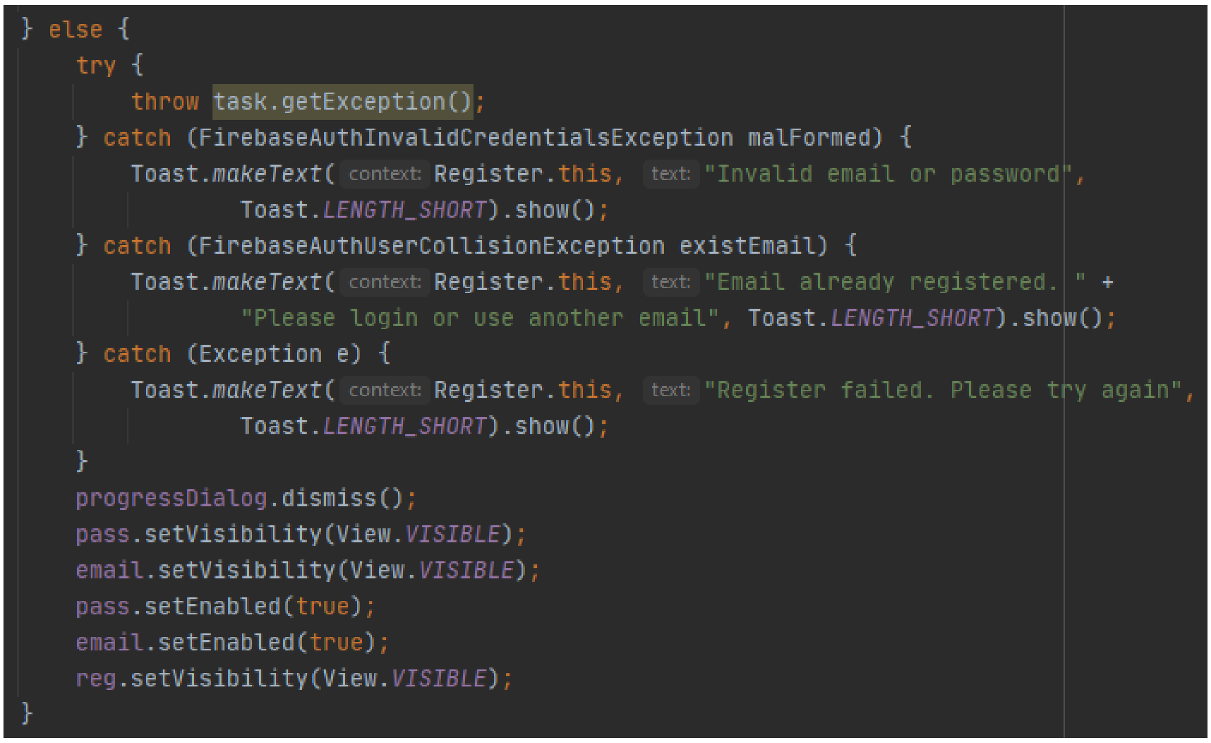
There exist two new *if* statements, the first *if* is to check whether the password length is less than 6 characters or not. If it is less than 6 characters, it will show an error saying that the password must be at least 6 characters long. The second *if* is to check whether the re-enter password field text is the same as the password field text. If it is different, the user is prompted to re-check the entered password and try registering the account again.



```java
progressDialog.setMessage("Registering...");
progressDialog.show();
pass.setVisibility(View.GONE);
email.setVisibility(View.GONE);
pass.setEnabled(false);
email.setEnabled(false);
reg.setVisibility(View.GONE);
mAuth.createUserWithEmailAndPassword(email_check, password_check)
        .addOnCompleteListener( activity: Register.this, new OnCompleteListener<AuthResult>() {
            @Override
            public void onComplete(@NonNull @org.jetbrains.annotations.NotNull Task<AuthResult> task) {
                if (task.isSuccessful()) {
                    Map<String, Object> new_user = new HashMap<>();
                    new_user.put("email", email_check);

                    FirebaseDatabase.getInstance().getReference( path: "Users")
                            .child(FirebaseAuth.getInstance().getCurrentUser().getUid())
                            .setValue(new_user).addOnCompleteListener( activity: Register.this, new OnCompleteListener<Void>() {
                        @Override
                        public void onComplete(@NonNull @org.jetbrains.annotations.NotNull Task<Void> task) {
                            if (task.isSuccessful()) {
                                Toast.makeText( context: Register.this,  text: email_check + " has been registered. " +
                                        "Please check your inbox to continue with the registration", Toast.LENGTH_SHORT).show();
                                FirebaseAuth.getInstance().getCurrentUser().sendEmailVerification();
                                mAuth.signOut();
                                finish();
                            } else {
                                Toast.makeText( context: Register.this,  text: "Failed to register " + email_check +
                                        ". Please try again", Toast.LENGTH_SHORT).show();
                            }
                            progressDialog.dismiss();
                            pass.setVisibility(View.VISIBLE);
                            email.setVisibility(View.VISIBLE);
                            pass.setEnabled(true);
                            email.setEnabled(true);
                            reg.setVisibility(View.VISIBLE);
                        }
                    });
```

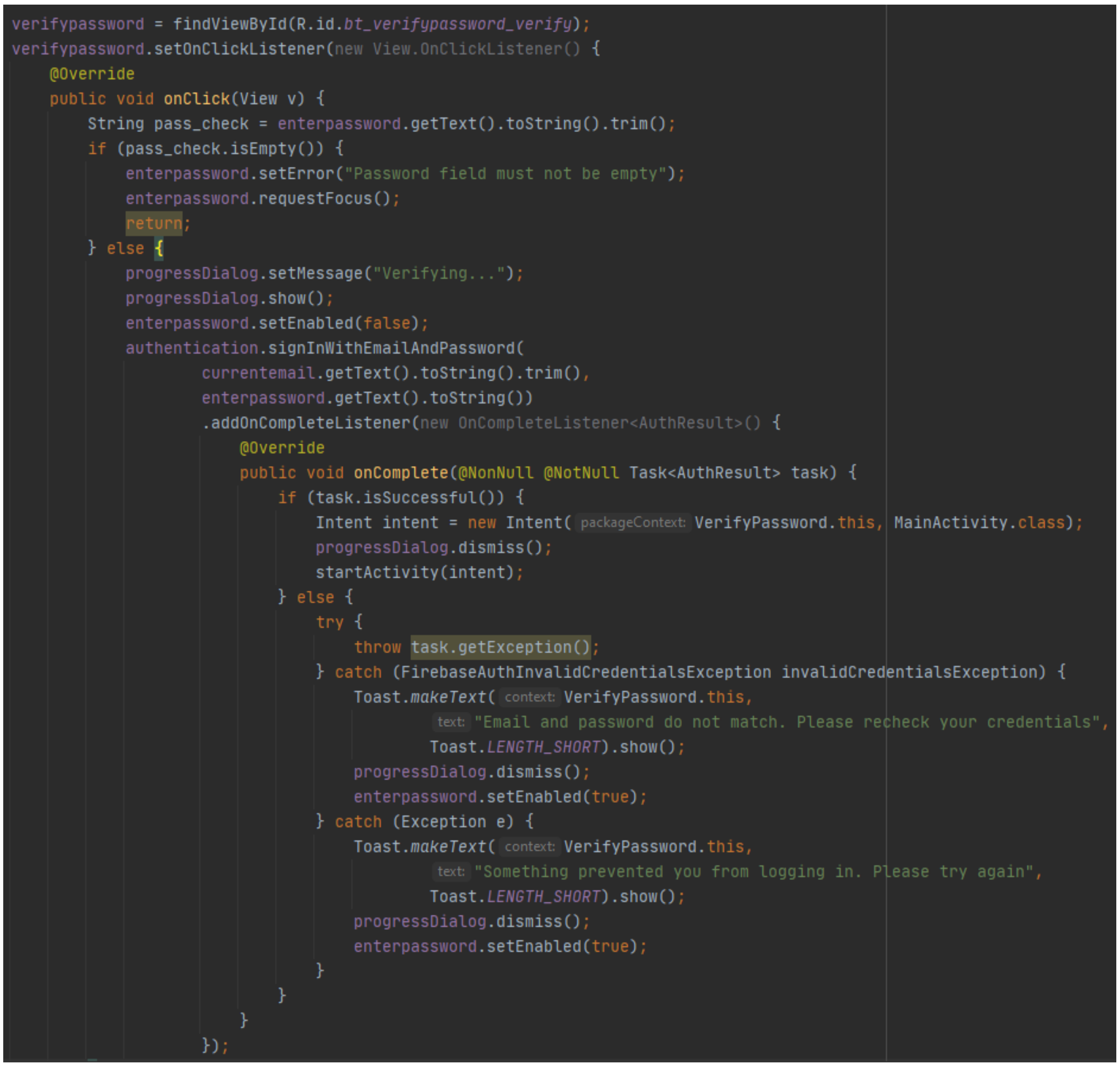
mAuth is the *FirebaseAuth* object that has been initialized before in the overridden *onCreate()* method. Instead of using the *signInWithEmailAndPassword()* method, the method now uses the *createUserWithEmailAndPassword()* method, which will create a new user based on the information that the user has entered before in the text fields. After the credentials have been stored into the database, the method will then create a new path in the database, which is the user ID generated automatically by Firebase. It will then push a single key-value pair that is the email address. There will be a message showing that the user needs to verify their email before they can continue using their new account after the process has been successful.

If the registering process failed, the method would catch the exceptions thrown by the registering process.



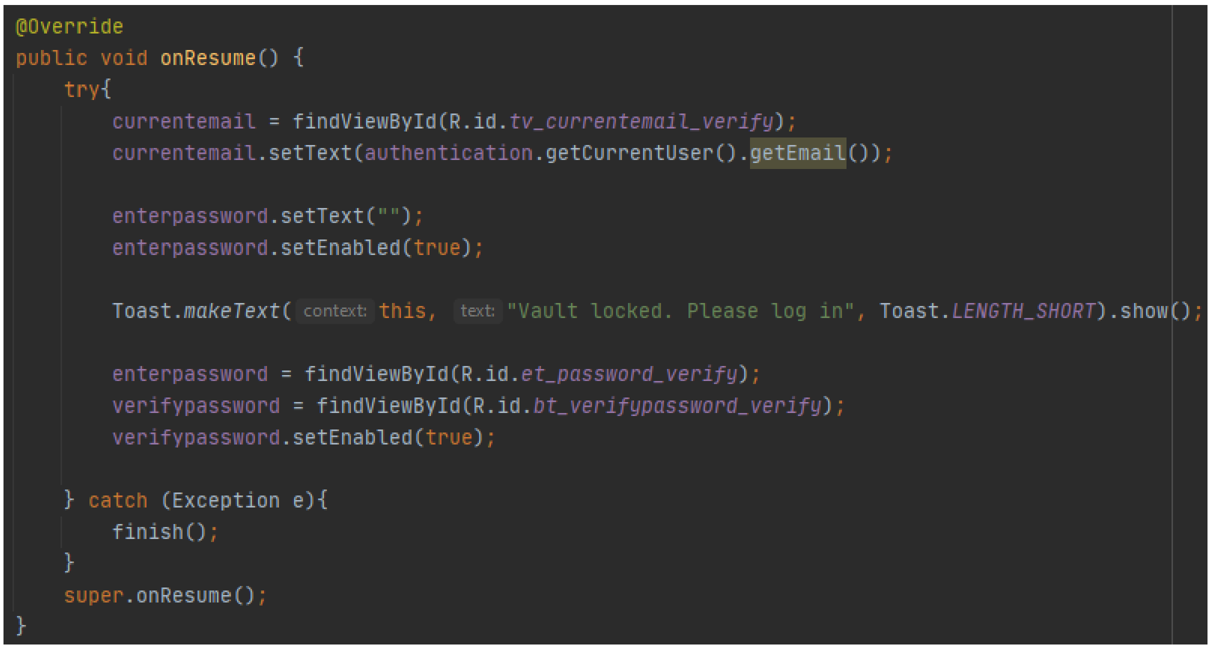
```
} else {
    try {
        throw task.getException();
    } catch (FirebaseAuthInvalidCredentialsException malFormed) {
        Toast.makeText( context: Register.this,  text: "Invalid email or password",
                Toast.LENGTH_SHORT).show();
    } catch (FirebaseAuthUserCollisionException existEmail) {
        Toast.makeText( context: Register.this,  text: "Email already registered. " +
                "Please login or use another email", Toast.LENGTH_SHORT).show();
    } catch (Exception e) {
        Toast.makeText( context: Register.this,  text: "Register failed. Please try again",
                Toast.LENGTH_SHORT).show();
    }
    progressDialog.dismiss();
    pass.setVisibility(View.VISIBLE);
    email.setVisibility(View.VISIBLE);
    pass.setEnabled(true);
    email.setEnabled(true);
    reg.setVisibility(View.VISIBLE);
}
```

c. Verify Password Class



```
verifypassword = findViewById(R.id.bt_verifypassword_verify);
verifypassword.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String pass_check = enterpassword.getText().toString().trim();
        if (pass_check.isEmpty()) {
            enterpassword.setError("Password field must not be empty");
            enterpassword.requestFocus();
            return;
        } else {
            progressDialog.setMessage("Verifying...");
            progressDialog.show();
            enterpassword.setEnabled(false);
            authentication.signInWithEmailAndPassword(
                    currentemail.getText().toString().trim(),
                    enterpassword.getText().toString())
                    .addOnCompleteListener(new OnCompleteListener<AuthResult>() {
                        @Override
                        public void onComplete(@NonNull @NotNull Task<AuthResult> task) {
                            if (task.isSuccessful()) {
                                Intent intent = new Intent( packageContext: VerifyPassword.this, MainActivity.class);
                                progressDialog.dismiss();
                                startActivity(intent);
                            } else {
                                try {
                                    throw task.getException();
                                } catch (FirebaseAuthInvalidCredentialsException invalidCredentialsException) {
                                    Toast.makeText( context: VerifyPassword.this,
                                            text: "Email and password do not match. Please recheck your credentials",
                                            Toast.LENGTH_SHORT).show();
                                    progressDialog.dismiss();
                                    enterpassword.setEnabled(true);
                                } catch (Exception e) {
                                    Toast.makeText( context: VerifyPassword.this,
                                            text: "Something prevented you from logging in. Please try again",
                                            Toast.LENGTH_SHORT).show();
                                    progressDialog.dismiss();
                                    enterpassword.setEnabled(true);
                                }
                            }
                        }
                    });
```

The *Verify Password* class is an activity class where user needs to re-enter their password in order to access their password vault. This is an added security to keep their password safe. When the user has successfully re-entered their password and logged into the application, this activity will go a paused state. When the vault has timed out, this activity will be resumed and the *onResume()* method will be called.

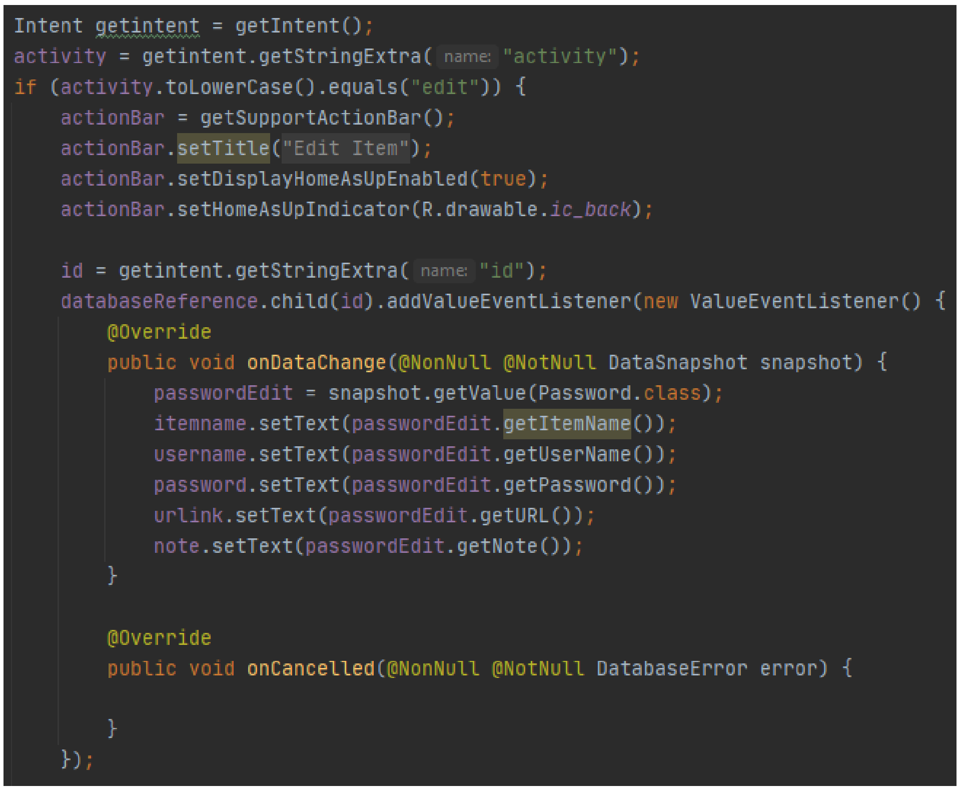


```
@Override
public void onResume() {
    try{
        currentemail = findViewById(R.id.tv_currentemail_verify);
        currentemail.setText(authentication.getCurrentUser().getEmail());

        enterpassword.setText("");
        enterpassword.setEnabled(true);

        Toast.makeText( context: this,  text: "Vault locked. Please log in", Toast.LENGTH_SHORT).show();

        enterpassword = findViewById(R.id.et_password_verify);
        verifypassword = findViewById(R.id.bt_verifypassword_verify);
        verifypassword.setEnabled(true);

    } catch (Exception e){
        finish();
    }
    super.onResume();
}
```

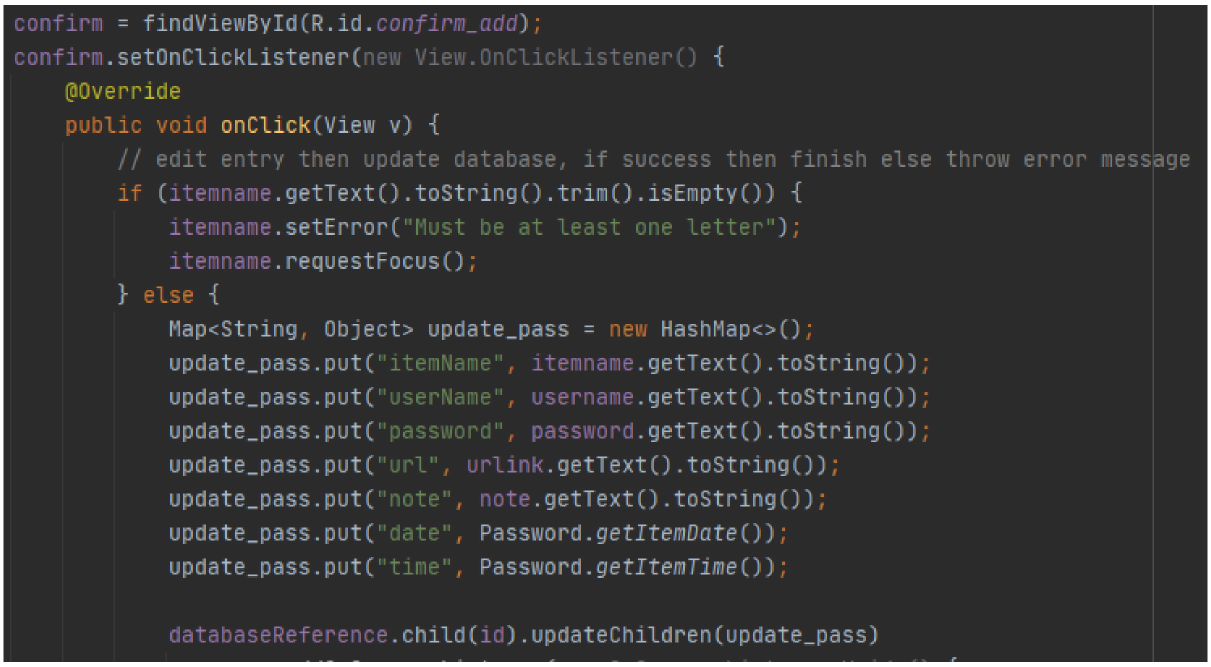
The *try catch* block here is to prevent the application from crashing if the user is signing out from the application. Because this activity is never finished, it will stay on the background. When the user has exited from the *Main Activity*, the application will go into a new *Login* activity. And when the user presses the back button, the application will go into this activity and will crash because the *FirebaseAuth.getCurrentUser().getEmail()* call will return a *NullPointerException*, because no user is currently logged in to the application. As for why it will go to this activity when user presses the back button, it is because this activity always stays on the bottommost stack of the opened activities. When the user has signed out, a the *Main Activity* activity will be terminated and a new activity *Login* will be started, while still leaving the *Verify Password* activity on the bottom of the stack. When the user presses the back button, it will finish the *Login* activity and goes back to the *Verify Password* activity, thus calling the *onResume()* method with no user info (email) saved in the application.

d. Password Add/Edit Class

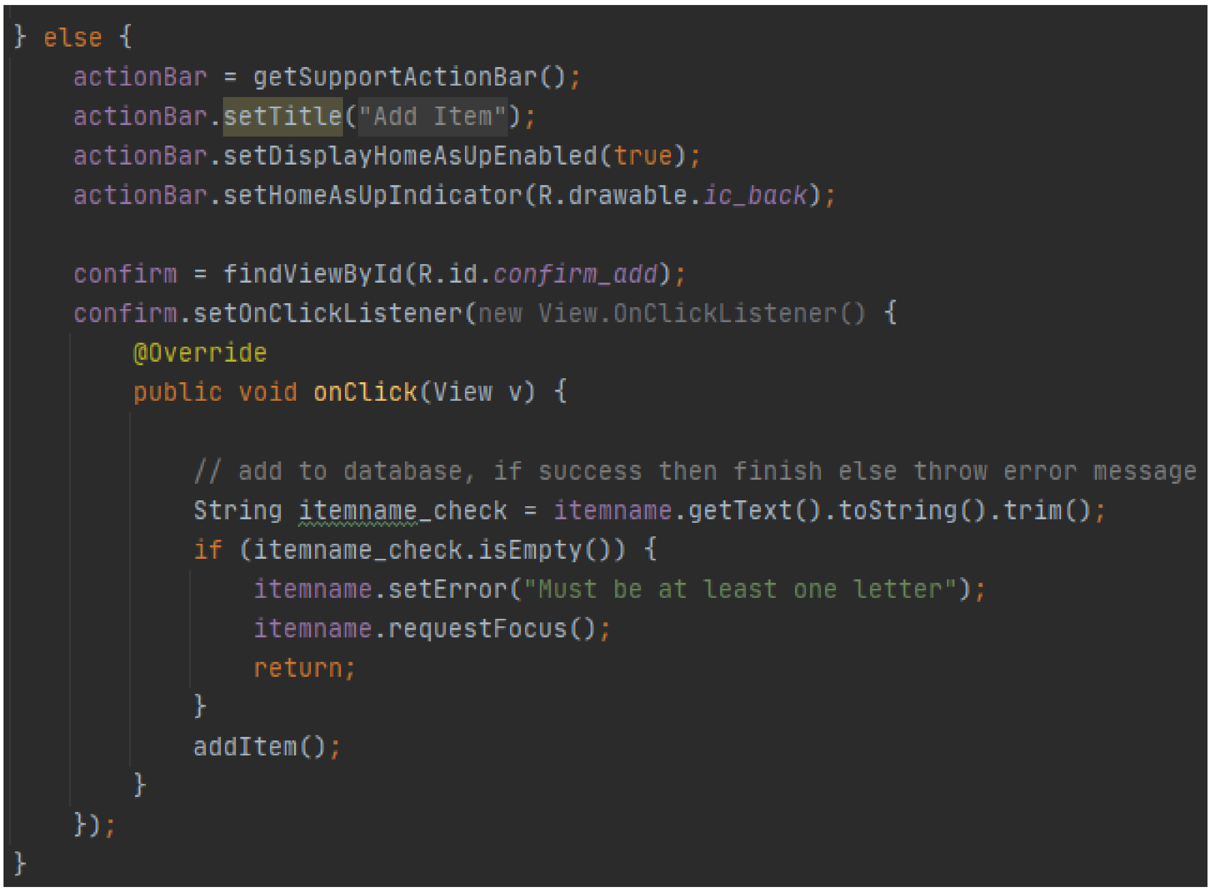
This class acts as the activity for add and edit entries.


```java
Intent getintent = getIntent();
activity = getintent.getStringExtra( name: "activity");
if (activity.toLowerCase().equals("edit")) {
    actionBar = getSupportActionBar();
    actionBar.setTitle("Edit Item");
    actionBar.setDisplayHomeAsUpEnabled(true);
    actionBar.setHomeAsUpIndicator(R.drawable.ic_back);

    id = getintent.getStringExtra( name: "id");
    databaseReference.child(id).addValueEventListener(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull @NotNull DataSnapshot snapshot) {
            passwordEdit = snapshot.getValue(Password.class);
            itemname.setText(passwordEdit.getItemName());
            username.setText(passwordEdit.getUserName());
            password.setText(passwordEdit.getPassword());
            urlink.setText(passwordEdit.getURL());
            note.setText(passwordEdit.getNote());
        }

        @Override
        public void onCancelled(@NonNull @NotNull DatabaseError error) {

        }
    });
```

This is the *if* statement that checks whether the activity is an edit or a new entry activity. If it is an edit entry, it will behave as an edit activity. Instead of leaving the fields empty, it will set them according to their respective fields. Item name field will be filled by the item name retrieved from the database, User name field will also be filled by the user name that have been retrieved from the database, and so on.

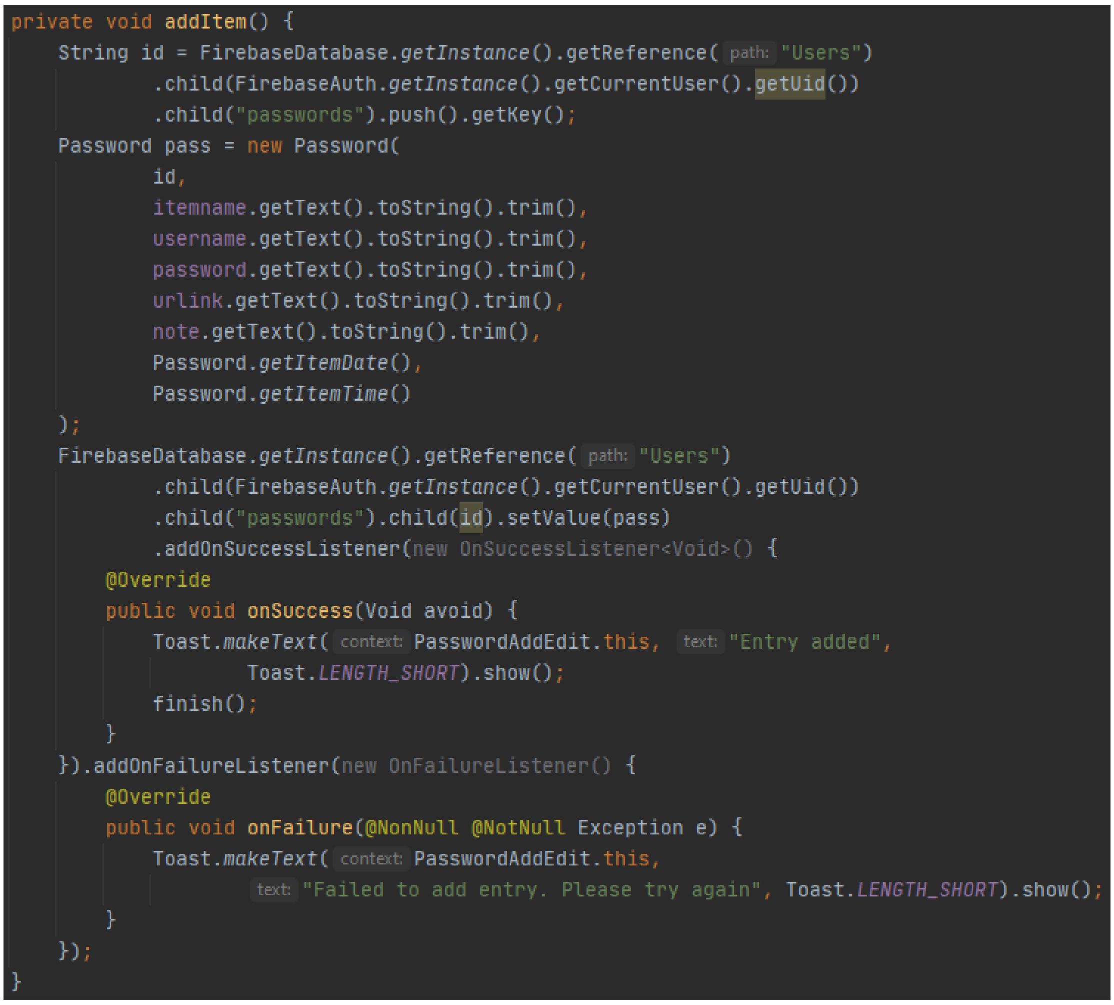

```java
confirm = findViewById(R.id.confirm_add);
confirm.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // edit entry then update database, if success then finish else throw error message
        if (itemname.getText().toString().trim().isEmpty()) {
            itemname.setError("Must be at least one letter");
            itemname.requestFocus();
        } else {
            Map<String, Object> update_pass = new HashMap<>();
            update_pass.put("itemName", itemname.getText().toString());
            update_pass.put("userName", username.getText().toString());
            update_pass.put("password", password.getText().toString());
            update_pass.put("url", urlink.getText().toString());
            update_pass.put("note", note.getText().toString());
            update_pass.put("date", Password.getItemDate());
            update_pass.put("time", Password.getItemTime());

            databaseReference.child(id).updateChildren(update_pass)
```

After the user has been satisfied with the changes and clicks the *confirm* button, it will then update the existing entry that have previously been added in the database.



```
} else {
    actionBar = getSupportActionBar();
    actionBar.setTitle("Add Item");
    actionBar.setDisplayHomeAsUpEnabled(true);
    actionBar.setHomeAsUpIndicator(R.drawable.ic_back);

    confirm = findViewById(R.id.confirm_add);
    confirm.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {

            // add to database, if success then finish else throw error message
            String itemname_check = itemname.getText().toString().trim();
            if (itemname_check.isEmpty()) {
                itemname.setError("Must be at least one letter");
                itemname.requestFocus();
                return;
            }
            addItem();
        }
    });
}
```

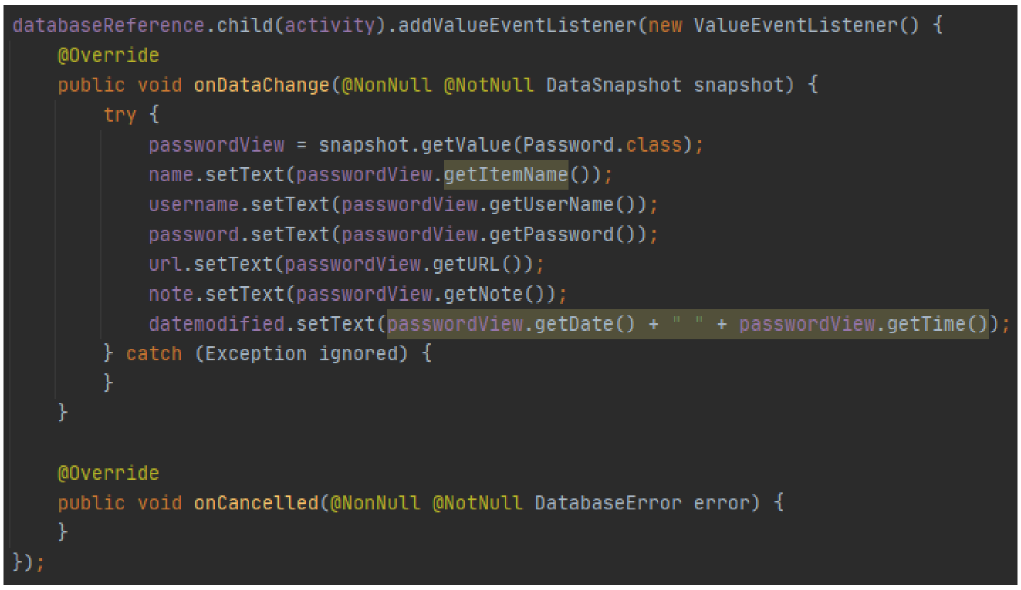
If the current activity is a new entry activity, then the activity will behave as a new entry activity with all the fields emptied.



```
private void addItem() {
    String id = FirebaseDatabase.getInstance().getReference( path: "Users")
            .child(FirebaseAuth.getInstance().getCurrentUser().getUid())
            .child("passwords").push().getKey();
    Password pass = new Password(
            id,
            itemname.getText().toString().trim(),
            username.getText().toString().trim(),
            password.getText().toString().trim(),
            urlink.getText().toString().trim(),
            note.getText().toString().trim(),
            Password.getItemDate(),
            Password.getItemTime()
    );
    FirebaseDatabase.getInstance().getReference( path: "Users")
            .child(FirebaseAuth.getInstance().getCurrentUser().getUid())
            .child("passwords").child(id).setValue(pass)
            .addOnSuccessListener(new OnSuccessListener<Void>() {
        @Override
        public void onSuccess(Void avoid) {
            Toast.makeText( context: PasswordAddEdit.this,  text: "Entry added",
                    Toast.LENGTH_SHORT).show();
            finish();
        }
    }).addOnFailureListener(new OnFailureListener() {
        @Override
        public void onFailure(@NonNull @NotNull Exception e) {
            Toast.makeText( context: PasswordAddEdit.this,
                     text: "Failed to add entry. Please try again", Toast.LENGTH_SHORT).show();
        }
    });
}
```

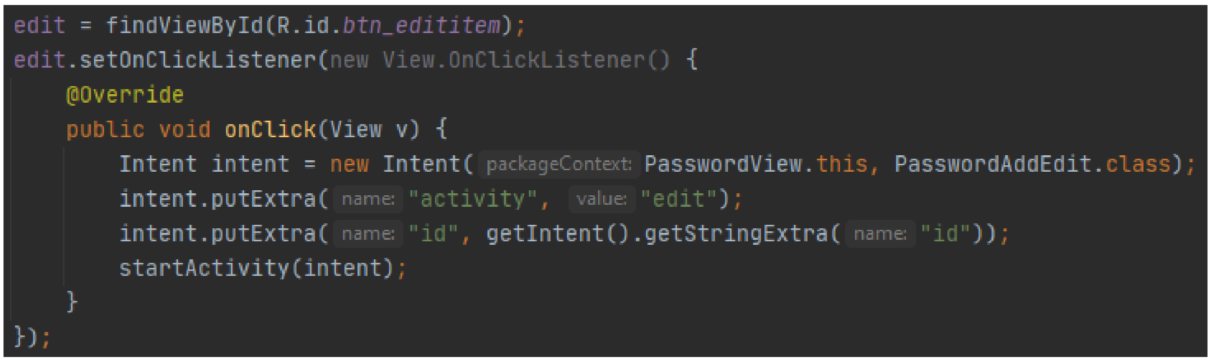
This is the *addItem()* method instructions that was called in the *else* statement.

e. Password View Class



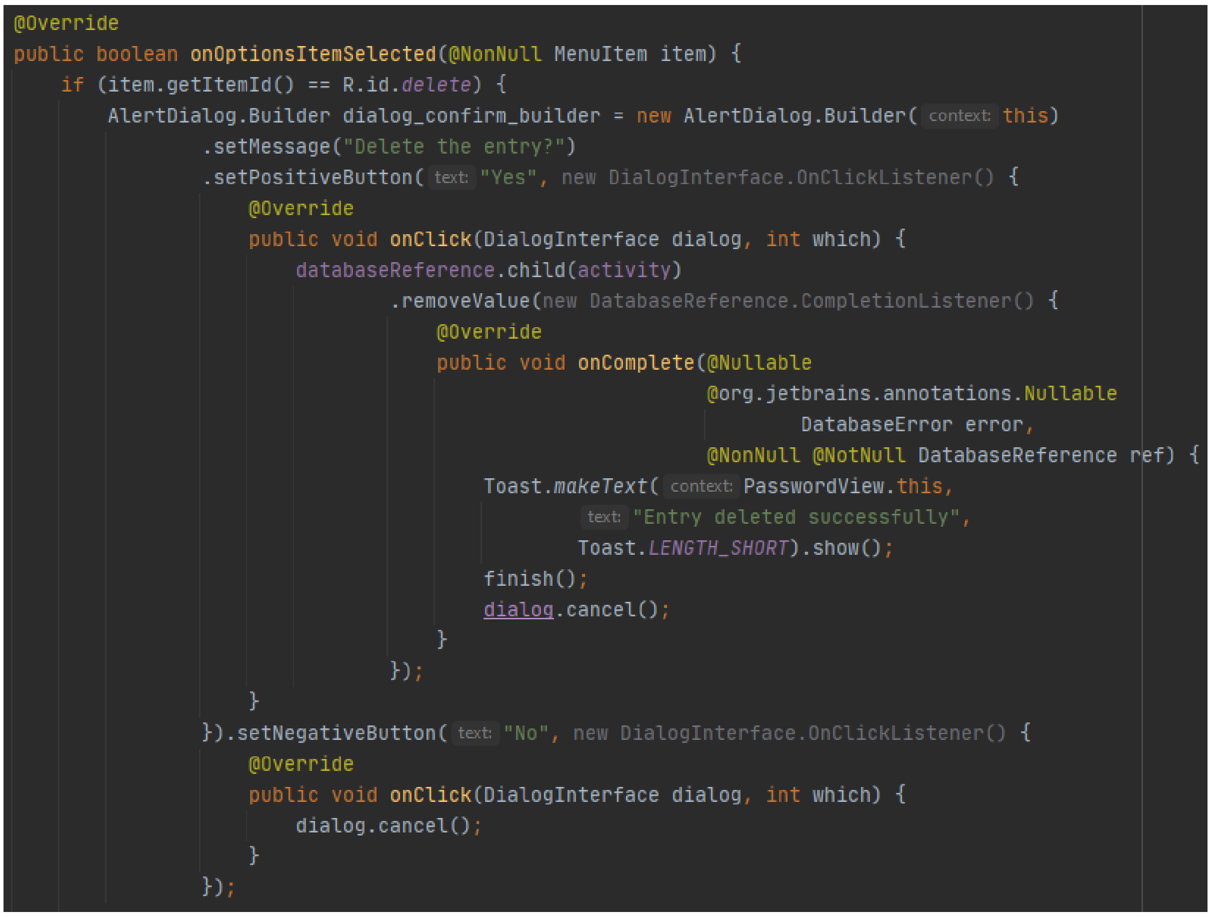
```
databaseReference.child(activity).addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull @NotNull DataSnapshot snapshot) {
        try {
            passwordView = snapshot.getValue(Password.class);
            name.setText(passwordView.getItemName());
            username.setText(passwordView.getUserName());
            password.setText(passwordView.getPassword());
            url.setText(passwordView.getURL());
            note.setText(passwordView.getNote());
            datemodified.setText(passwordView.getDate() + " " + passwordView.getTime());
        } catch (Exception ignored) {
        }
    }

    @Override
    public void onCancelled(@NonNull @NotNull DatabaseError error) {
    }
});
```

This class acts as the activity to view the entries that the user has entered before. When the user clicks on an entry, they will be redirected into this activity that will show all the information that have been entered into the entry. As an addition, it will also show when the entry was last modified by the user.



```
edit = findViewById(R.id.btn_edititem);
edit.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent( packageContext: PasswordView.this, PasswordAddEdit.class);
        intent.putExtra( name: "activity",  value: "edit");
        intent.putExtra( name: "id", getIntent().getStringExtra( name: "id"));
        startActivity(intent);
    }
});
```

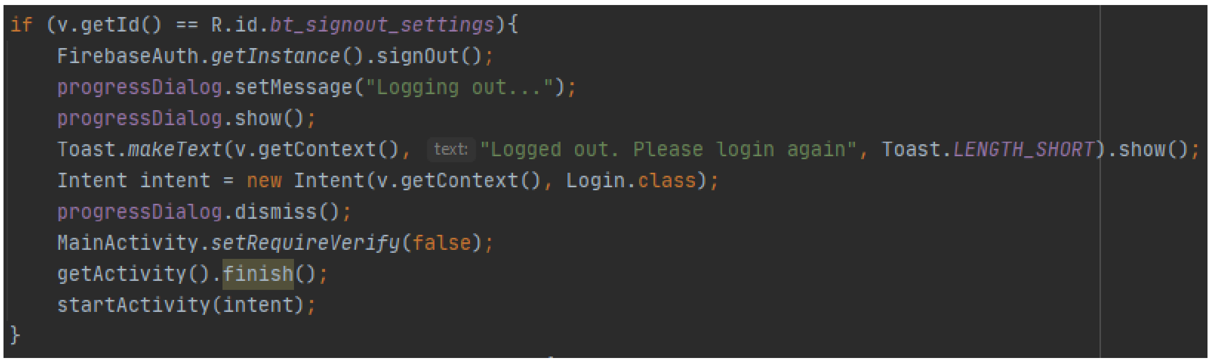
The user is given the option to edit the entry. If the user clicks on the edit button, they will be redirected to the *Password Add/Edit* activity where they can freely edit it.



```
@Override
public boolean onOptionsItemSelected(@NonNull MenuItem item) {
    if (item.getItemId() == R.id.delete) {
        AlertDialog.Builder dialog_confirm_builder = new AlertDialog.Builder( context: this)
                .setMessage("Delete the entry?")
                .setPositiveButton( text: "Yes", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        databaseReference.child(activity)
                                .removeValue(new DatabaseReference.CompletionListener() {
                                    @Override
                                    public void onComplete(@Nullable
                                                           @org.jetbrains.annotations.Nullable
                                                                   DatabaseError error,
                                                           @NonNull @NotNull DatabaseReference ref) {
                                        Toast.makeText( context: PasswordView.this,
                                                 text: "Entry deleted successfully",
                                                Toast.LENGTH_SHORT).show();
                                        finish();
                                        dialog.cancel();
                                    }
                                });
                    }
                }).setNegativeButton( text: "No", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        dialog.cancel();
                    }
                });
```

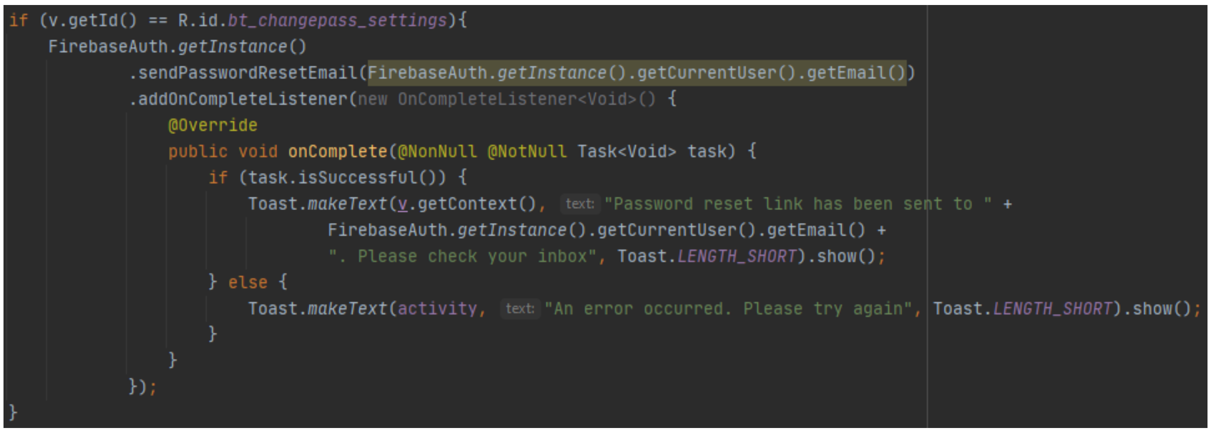
Other than the ability to edit the entry, the user is given the option to delete the entry. The code snippet above shows how the deletion process goes. Before deleting the entry, a confirmation dialog will be shown to the user to make sure that the user really wants to delete the entry. If they changed their mind and decided to cancel the process, the application will do nothing, else if the user is sure about deleting the entry, then the application will send a request to Firebase telling it that the entry should be deleted. Variable *activity* is the holder for the entry's ID.

f. Account Fragment Class



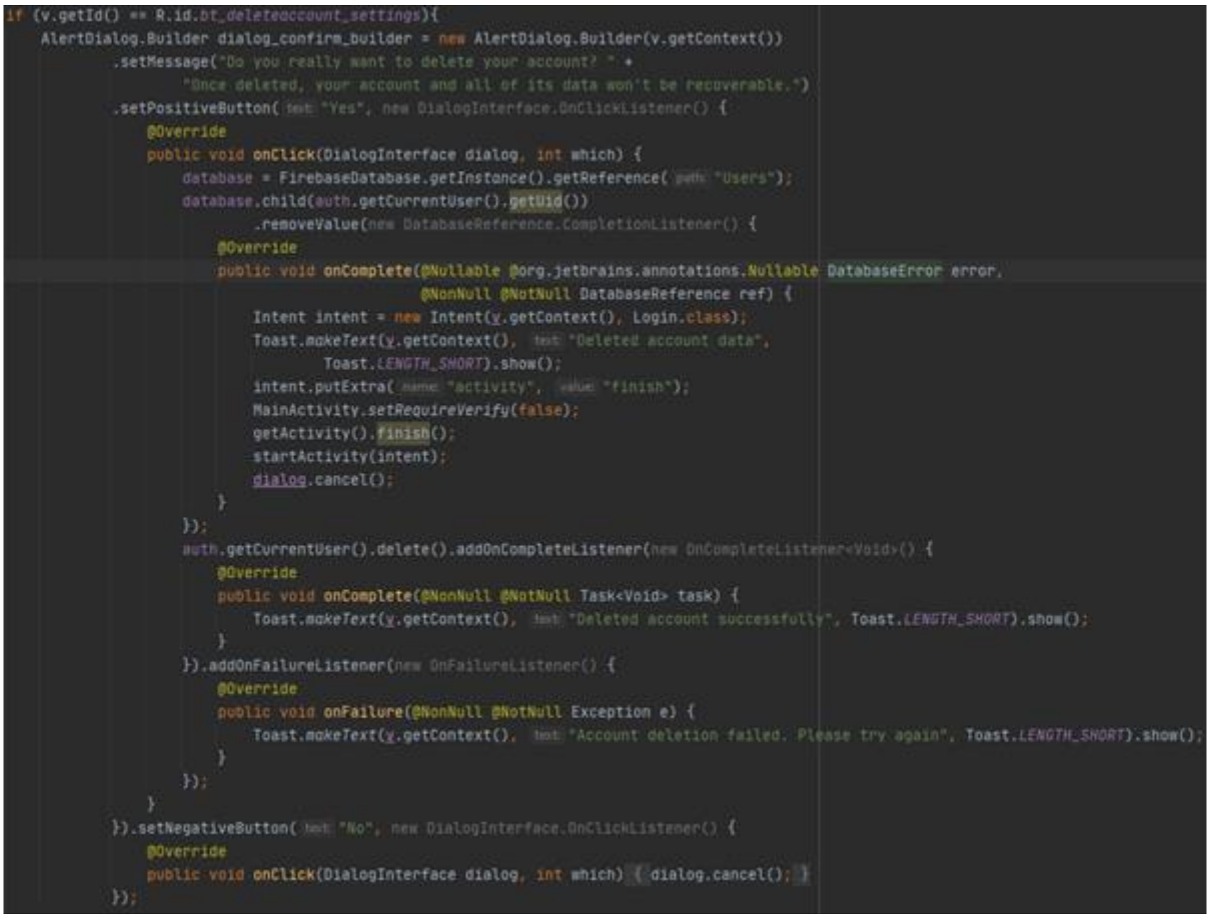
```
if (v.getId() == R.id.bt_signout_settings){
    FirebaseAuth.getInstance().signOut();
    progressDialog.setMessage("Logging out...");
    progressDialog.show();
    Toast.makeText(v.getContext(),  text: "Logged out. Please login again", Toast.LENGTH_SHORT).show();
    Intent intent = new Intent(v.getContext(), Login.class);
    progressDialog.dismiss();
    MainActivity.setRequireVerify(false);
    getActivity().finish();
    startActivity(intent);
}
```

The first option that the user can choose upon opening the *Account* activity is the *sign out* option. This option, if clicked, logs the user out of the application. The application will then prompt the next user to input their credentials to log into the app.



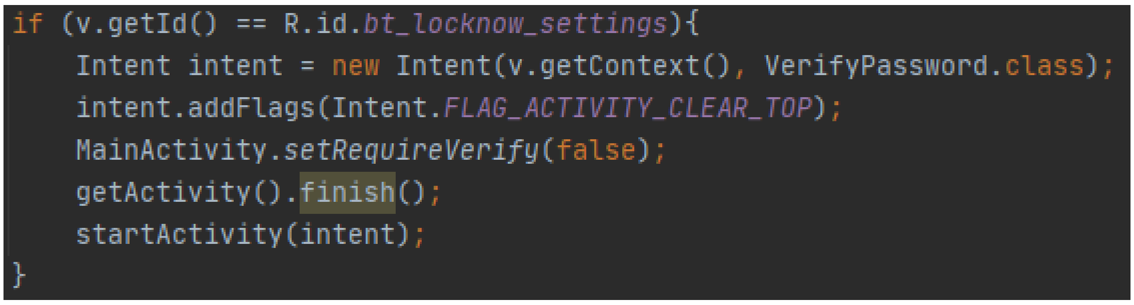
```
if (v.getId() == R.id.bt_changepass_settings){
    FirebaseAuth.getInstance()
            .sendPasswordResetEmail(FirebaseAuth.getInstance().getCurrentUser().getEmail())
            .addOnCompleteListener(new OnCompleteListener<Void>() {
                @Override
                public void onComplete(@NonNull @NotNull Task<Void> task) {
                    if (task.isSuccessful()) {
                        Toast.makeText(v.getContext(),  text: "Password reset link has been sent to " +
                                FirebaseAuth.getInstance().getCurrentUser().getEmail() +
                                ". Please check your inbox", Toast.LENGTH_SHORT).show();
                    } else {
                        Toast.makeText(activity,  text: "An error occurred. Please try again", Toast.LENGTH_SHORT).show();
                    }
                }
            });
}
```

The second option is the *change password* option. User can change their password by clicking on this button. Once clicked, a password reset confirmation link will be sent to the user’s email inbox. From there, the user can open the password reset link and reset their password.



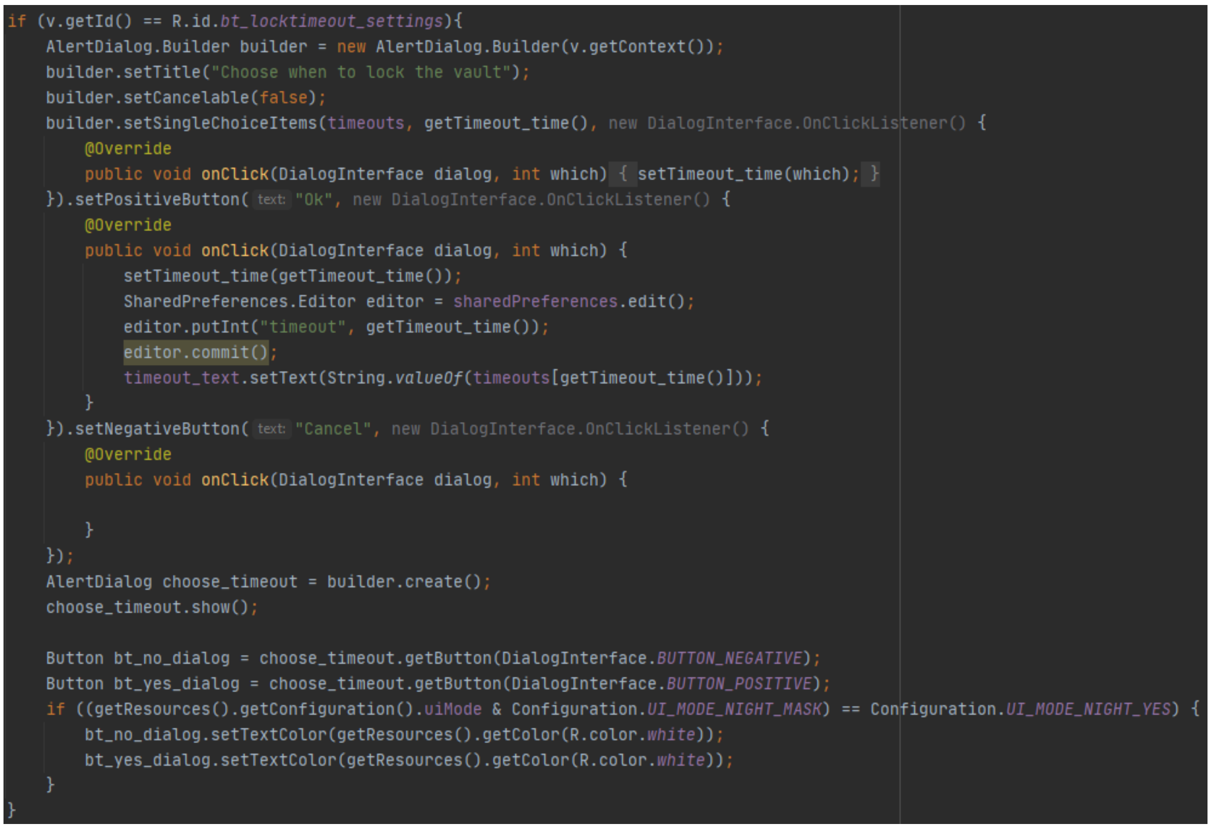
```java
if (v.getId() == R.id.bt_deleteaccount_settings){
    AlertDialog.Builder dialog_confirm_builder = new AlertDialog.Builder(v.getContext())
            .setMessage("Do you really want to delete your account? " +
                    "Once deleted, your account and all of its data won't be recoverable.")
            .setPositiveButton( text: "Yes", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    database = FirebaseDatabase.getInstance().getReference( path: "Users");
                    database.child(auth.getCurrentUser().getUid())
                            .removeValue(new DatabaseReference.CompletionListener() {
                        @Override
                        public void onComplete(@Nullable @org.jetbrains.annotations.Nullable DatabaseError error,
                                               @NonNull @NotNull DatabaseReference ref) {
                            Intent intent = new Intent(v.getContext(), Login.class);
                            Toast.makeText(v.getContext(), text: "Deleted account data",
                                    Toast.LENGTH_SHORT).show();
                            intent.putExtra( name: "activity", value: "finish");
                            MainActivity.setRequireVerify(false);
                            getActivity().finish();
                            startActivity(intent);
                            dialog.cancel();
                        }
                    });
                    auth.getCurrentUser().delete().addOnCompleteListener(new OnCompleteListener<Void>() {
                        @Override
                        public void onComplete(@NonNull @NotNull Task<Void> task) {
                            Toast.makeText(v.getContext(), text: "Deleted account successfully", Toast.LENGTH_SHORT).show();
                        }
                    }).addOnFailureListener(new OnFailureListener() {
                        @Override
                        public void onFailure(@NonNull @NotNull Exception e) {
                            Toast.makeText(v.getContext(), text: "Account deletion failed. Please try again", Toast.LENGTH_SHORT).show();
                        }
                    });
                }
            }).setNegativeButton( text: "No", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) { dialog.cancel(); }
            });
```

The third option is the *delete account* option. When this option is pressed, a confirmation dialog box will open to confirm that the user is sure to delete their account. If the user confirms it, then the application will request Firebase for account deletion. The process is quite simple. First, it will delete the path which is the user ID in the database by calling *FirebaseDatabase.getReference().child(getCurrentUser().getUID()).removeValue()*. After it has successfully deleted the path, it will then delete the account itself by calling *FirebaseAuth.getCurrentUser().delete().* After all of the process are completed and successful, the user will be redirected to the *Login* activity and a message will pop up saying that the account deletion was successful.



```java
if (v.getId() == R.id.bt_locknow_settings){
    Intent intent = new Intent(v.getContext(), VerifyPassword.class);
    intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
    MainActivity.setRequireVerify(false);
    getActivity().finish();
    startActivity(intent);
}
```

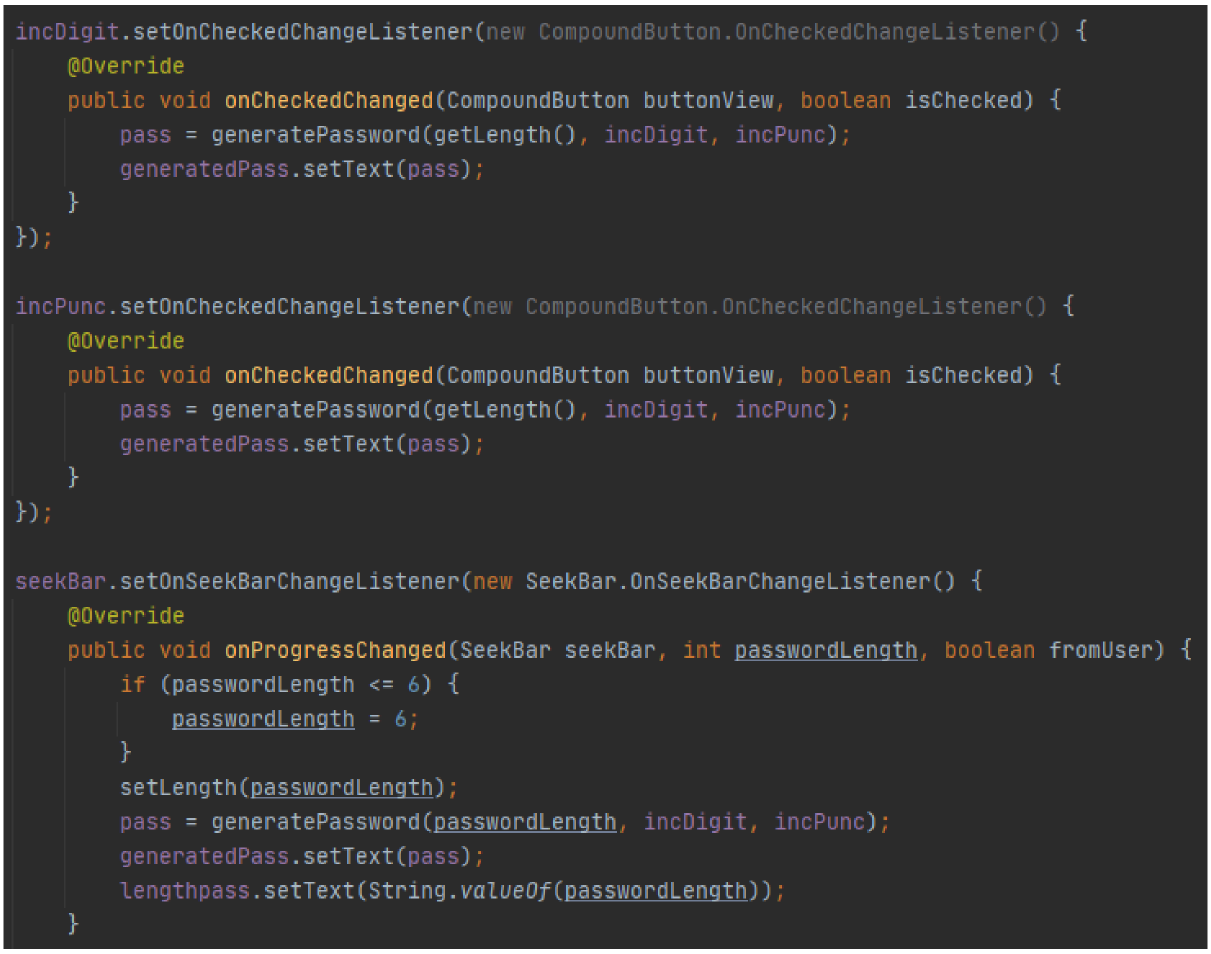
Aside from account control, user can also control some of the application's behavior. The first one is the *lock now* option. This option, when pressed, will lock the user out from their vault. This option can be useful when the user needs to quickly lock their vault.



```java
if (v.getId() == R.id.bt_locktimeout_settings){
    AlertDialog.Builder builder = new AlertDialog.Builder(v.getContext());
    builder.setTitle("Choose when to lock the vault");
    builder.setCancelable(false);
    builder.setSingleChoiceItems(timeouts, getTimeout_time(), new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) { setTimeout_time(which); }
    }).setPositiveButton( text: "Ok", new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            setTimeout_time(getTimeout_time());
            SharedPreferences.Editor editor = sharedPreferences.edit();
            editor.putInt("timeout", getTimeout_time());
            editor.commit();
            timeout_text.setText(String.valueOf(timeouts[getTimeout_time()]));
        }
    }).setNegativeButton( text: "Cancel", new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {

        }
    });
    AlertDialog choose_timeout = builder.create();
    choose_timeout.show();

    Button bt_no_dialog = choose_timeout.getButton(DialogInterface.BUTTON_NEGATIVE);
    Button bt_yes_dialog = choose_timeout.getButton(DialogInterface.BUTTON_POSITIVE);
    if ((getResources().getConfiguration().uiMode & Configuration.UI_MODE_NIGHT_MASK) == Configuration.UI_MODE_NIGHT_YES) {
        bt_no_dialog.setTextColor(getResources().getColor(R.color.white));
        bt_yes_dialog.setTextColor(getResources().getColor(R.color.white));
    }
}
```

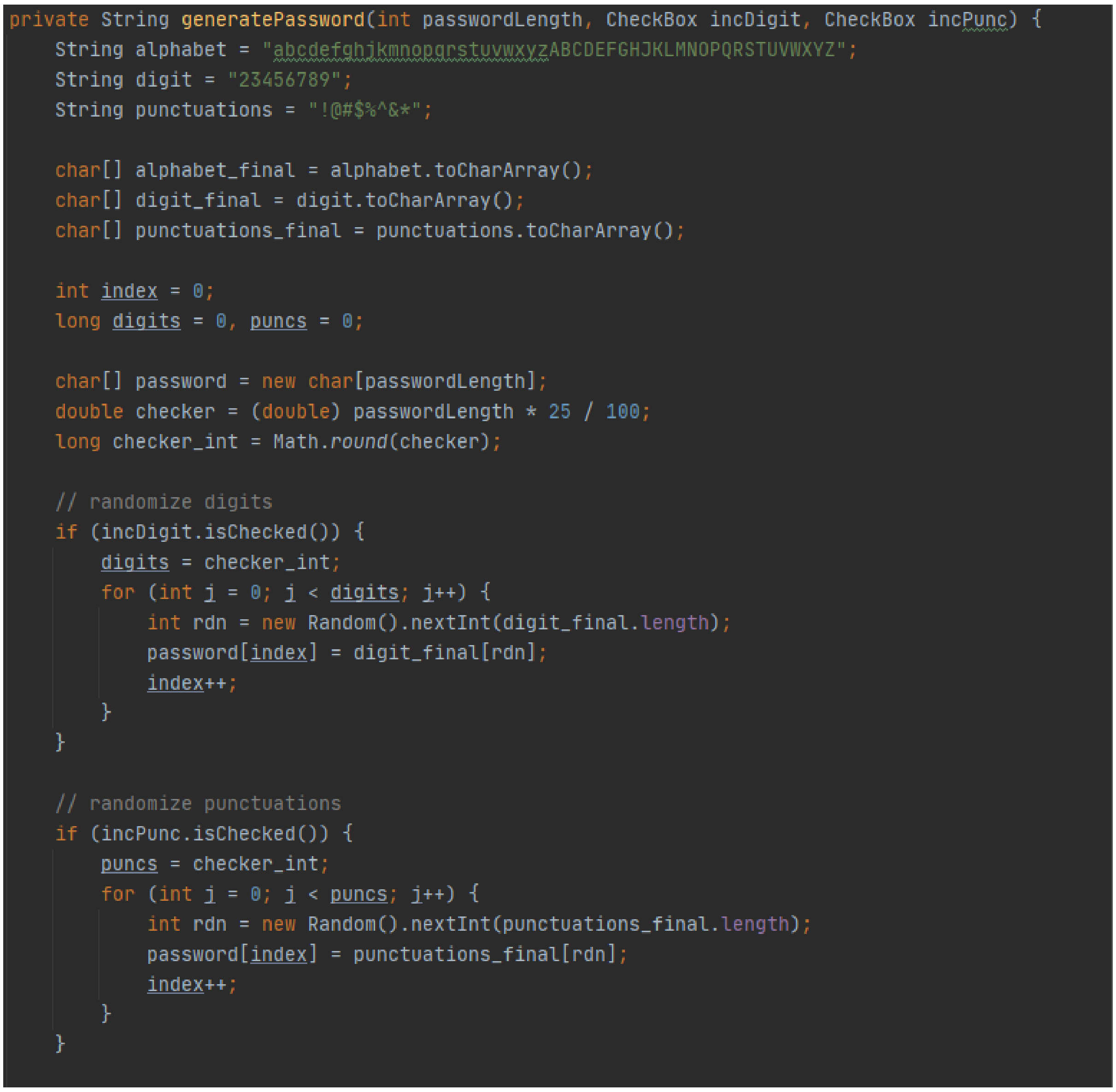
The last option that the user can choose is the *timeout* setting. This option, when clicked, will open a new pop-up dialog with a list of timeout options that are available for the user to choose. As stated in the features section above, there will be timeout options: 1 minute, 5 minutes, 10 minutes, 15 minutes, 20 minutes, 30 minutes, and immediate lock after exit. The *SharedPreferences* class is the class to save the current state of the application. When the user has chosen the timeout setting, it will then be saved into the app's internal data so that when the user later opens the app, they do not need to re-do their preferred timeout setting again.

g. Generator Fragment Class

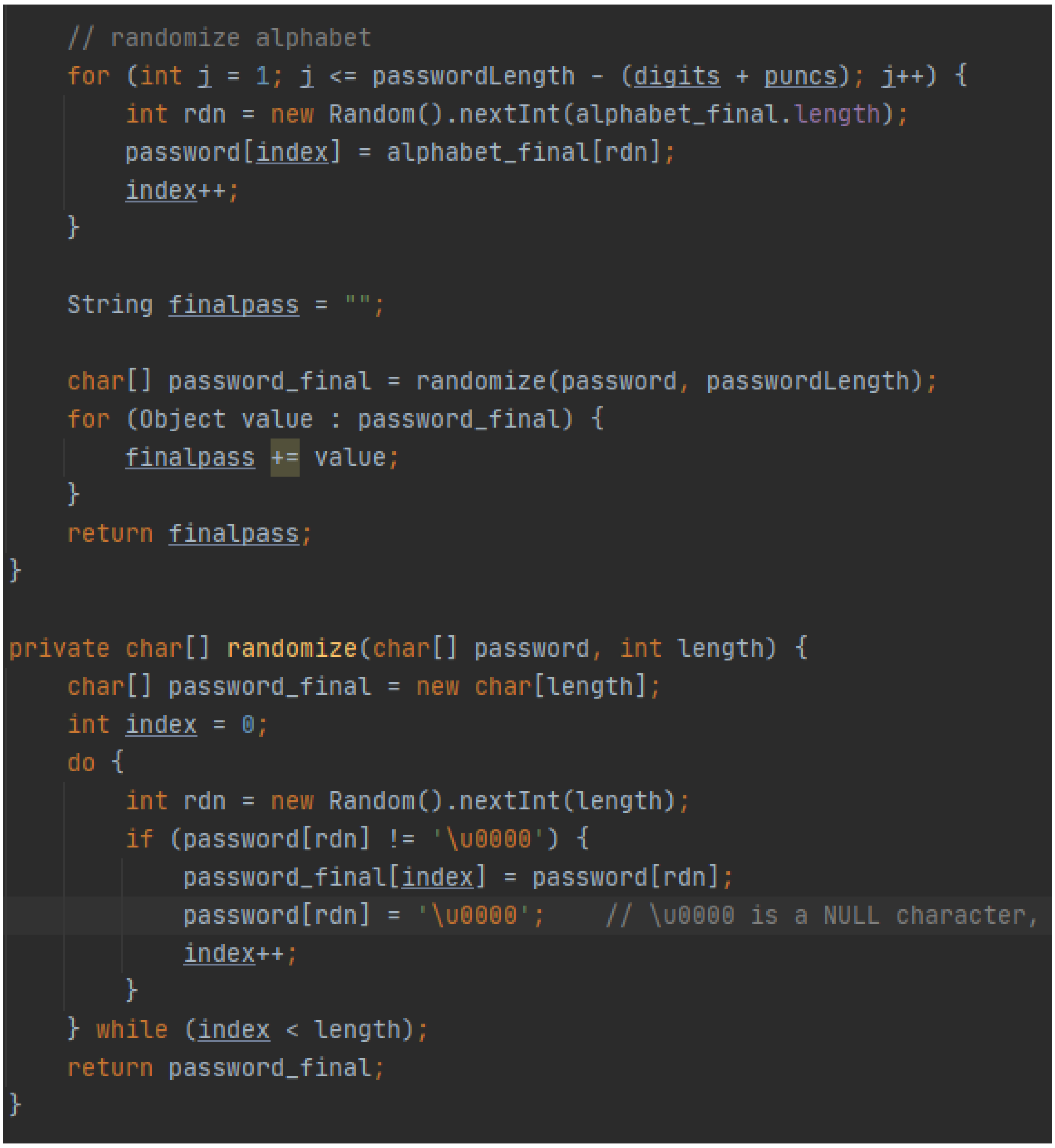


```
incDigit.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        pass = generatePassword(getLength(), incDigit, incPunc);
        generatedPass.setText(pass);
    }
});

incPunc.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        pass = generatePassword(getLength(), incDigit, incPunc);
        generatedPass.setText(pass);
    }
});

seekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
    @Override
    public void onProgressChanged(SeekBar seekBar, int passwordLength, boolean fromUser) {
        if (passwordLength <= 6) {
            passwordLength = 6;
        }
        setLength(passwordLength);
        pass = generatePassword(passwordLength, incDigit, incPunc);
        generatedPass.setText(pass);
        lengthpass.setText(String.valueOf(passwordLength));
    }
```

There are four options that the user can choose when generating a password, the first one is the length of the password, which is represented in a seek bar (a draggable bar with a dot as the cursor that can be moved left or right). The second one is the *include digit* option. This option is called *incDigit* in the code snippet above (an abbreviation of *include digit).* The third option is the *include punctuations* option. This option is called *incPunc* in the code snippet above (an abbreviation of *include punctuations).* The last one is the option to copy the generated password to the clipboard, so the user can paste it anywhere they want. The code snippet above shows the option for the *include digit*, *include* *punctuations*, and *password length* options. All of them will call *generatePassword()* method to generate the password. The method will return a *String*, which is the generated password.



```java
private String generatePassword(int passwordLength, CheckBox incDigit, CheckBox incPunc) {
    String alphabet = "abcdefghjkmnopqrstuvwxyzABCDEFGHJKLMNOPQRSTUVWXYZ";
    String digit = "23456789";
    String punctuations = "!@#$%^&*";

    char[] alphabet_final = alphabet.toCharArray();
    char[] digit_final = digit.toCharArray();
    char[] punctuations_final = punctuations.toCharArray();

    int index = 0;
    long digits = 0, puncs = 0;

    char[] password = new char[passwordLength];
    double checker = (double) passwordLength * 25 / 100;
    long checker_int = Math.round(checker);

    // randomize digits
    if (incDigit.isChecked()) {
        digits = checker_int;
        for (int j = 0; j < digits; j++) {
            int rdn = new Random().nextInt(digit_final.length);
            password[index] = digit_final[rdn];
            index++;
        }
    }

    // randomize punctuations
    if (incPunc.isChecked()) {
        puncs = checker_int;
        for (int j = 0; j < puncs; j++) {
            int rdn = new Random().nextInt(punctuations_final.length);
            password[index] = punctuations_final[rdn];
            index++;
        }
    }
```

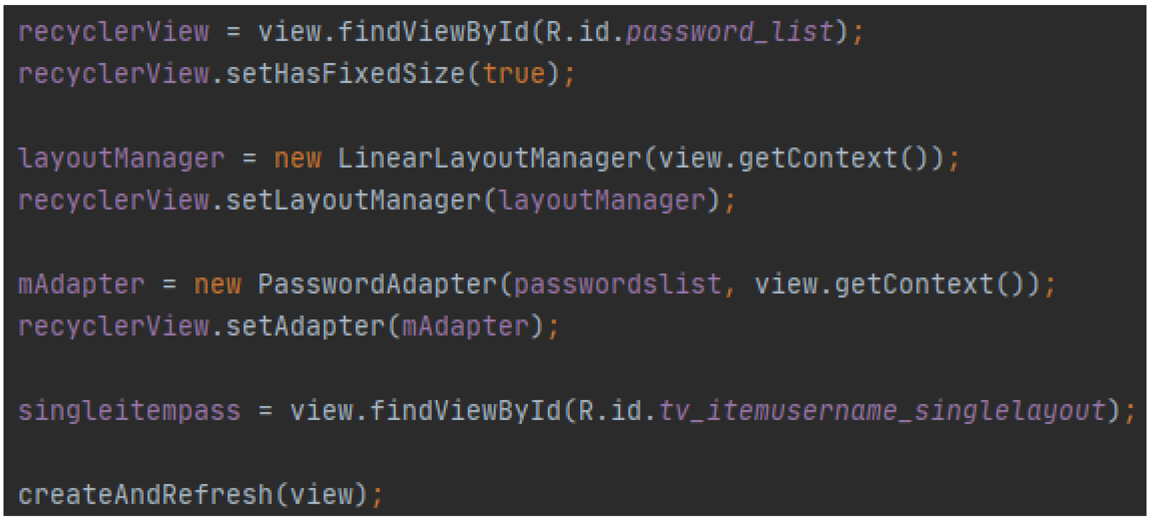


```
    // randomize alphabet
    for (int j = 1; j <= passwordLength - (digits + puncs); j++) {
        int rdn = new Random().nextInt(alphabet_final.length);
        password[index] = alphabet_final[rdn];
        index++;
    }

    String finalpass = "";

    char[] password_final = randomize(password, passwordLength);
    for (Object value : password_final) {
        finalpass += value;
    }
    return finalpass;
}

private char[] randomize(char[] password, int length) {
    char[] password_final = new char[length];
    int index = 0;
    do {
        int rdn = new Random().nextInt(length);
        if (password[rdn] != '\u0000') {
            password_final[index] = password[rdn];
            password[rdn] = '\u0000';    // \u0000 is a NULL character,
            index++;
        }
    } while (index < length);
    return password_final;
}
```

Both of the above's code snippets are the *generatePassword()* method, which generates a new password for the user. First it will check whether the *include digit* and or *include punctuations* are checked or not. If one or both of them are checked, then go to the *if* statement that will add the digits and or punctuations into the password array. It will add them sequentially. For example, the password length that the user want is 10. If the user checks both *include digits* and *include punctuations* options, first of all, it will add digits 25% of the password length into the temporary array. That means, the first index until the 2.5 ≈ 3th index will be filled by digits. The first 3 indexes will be filled by punctuations instead if the *include punctuations* option is checked and the *include digits* option is not checked. When both options are checked, the first 3 indexes will be filled by digits, and the next 3 indexes will be filled by punctuations. After that, it will fill the rest of the temporary array by a sequence of randomized lowercase and uppercase letters. Finally, it will then be randomized using the *private* method *randomize()* that will take the temporary array and the password length as the parameters. After it has successfully been randomized, it will then be converted into a *String* and will be returned to be then showed to the user.

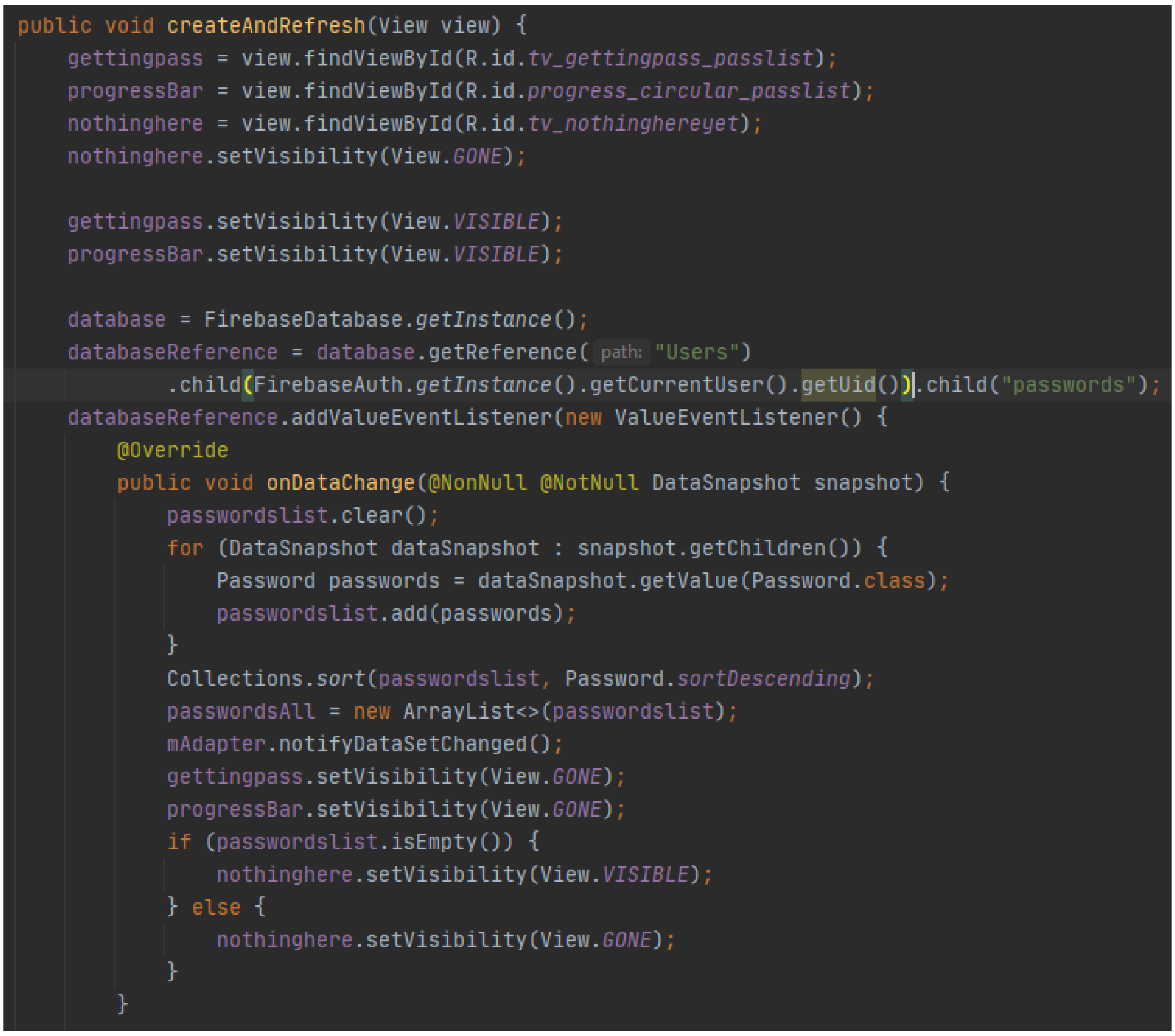
h. Password List Fragment Class

Of all the activities in this application, *Password List* can be considered "the most important one". It contains all of the entries that have been added by the user. From there, the user can search, edit, and delete the entries. Other than that, the user can also refresh the page by tapping on the *refresh* option on the dropdown menu, or by scrolling down the page. User can also directly log out from the application from the dropdown menu on the top right corner of the application.



```
recyclerView = view.findViewById(R.id.password_list);
recyclerView.setHasFixedSize(true);

layoutManager = new LinearLayoutManager(view.getContext());
recyclerView.setLayoutManager(layoutManager);

mAdapter = new PasswordAdapter(passwordslist, view.getContext());
recyclerView.setAdapter(mAdapter);

singleitempass = view.findViewById(R.id.tv_itemusername_singlelayout);

createAndRefresh(view);
```

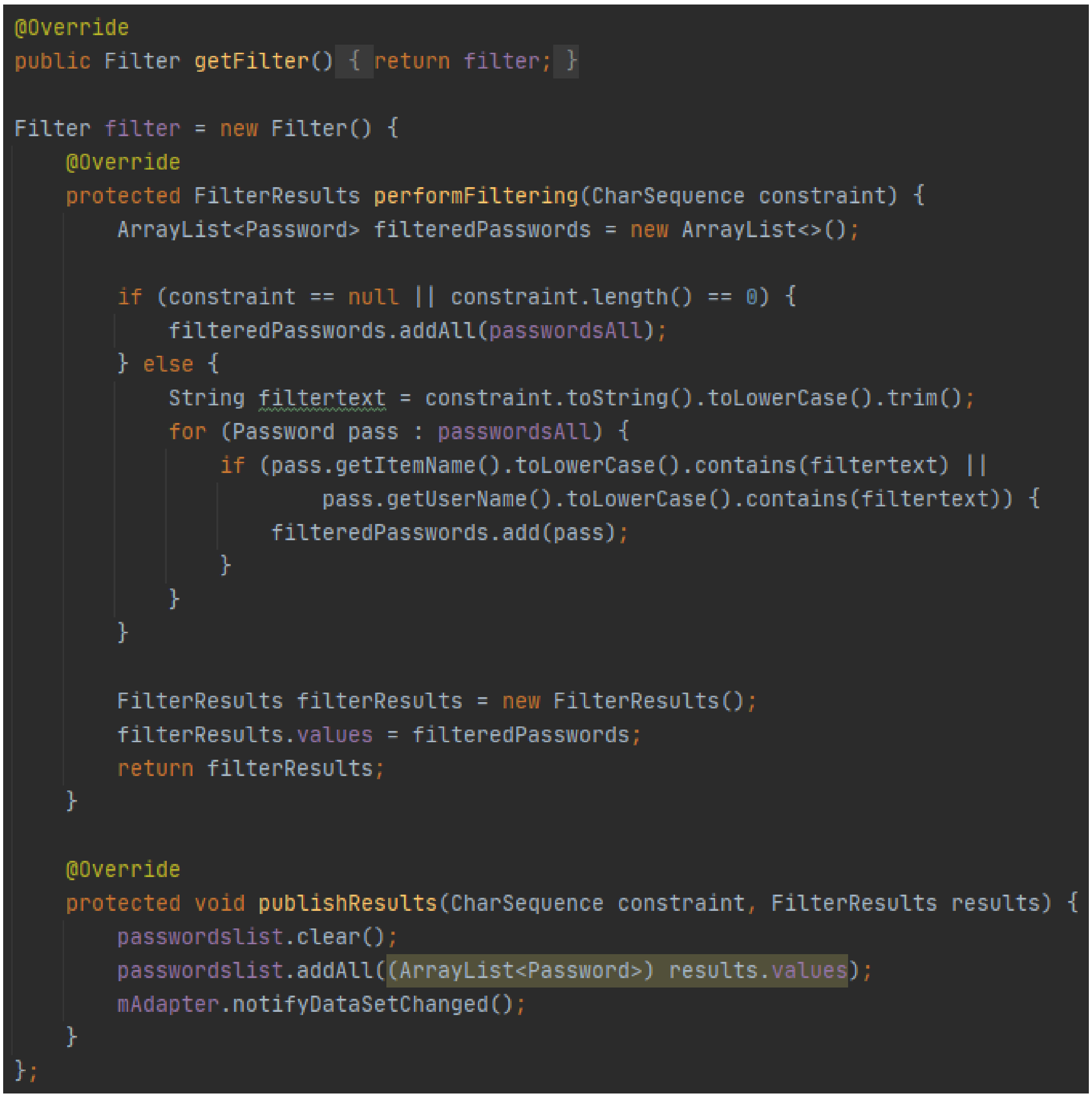
This code snippet above is where the recycler view gets initialized and set up with its adapter. The adapter will take care of the items that will later be shown in the view. Further explanation of the adapter will be discussed later in its respective section. After it has been initialized, it will call the *createAndRefresh()* method which will iterate through the database and add the entries to an *ArrayList*, from that *ArrayList*, the items can then be shown to the user.



```
public void createAndRefresh(View view) {
    gettingpass = view.findViewById(R.id.tv_gettingpass_passlist);
    progressBar = view.findViewById(R.id.progress_circular_passlist);
    nothinghere = view.findViewById(R.id.tv_nothinghereyet);
    nothinghere.setVisibility(View.GONE);

    gettingpass.setVisibility(View.VISIBLE);
    progressBar.setVisibility(View.VISIBLE);

    database = FirebaseDatabase.getInstance();
    databaseReference = database.getReference( path: "Users")
            .child(FirebaseAuth.getInstance().getCurrentUser().getUid()).child("passwords");
    databaseReference.addValueEventListener(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull @NotNull DataSnapshot snapshot) {
            passwordslist.clear();
            for (DataSnapshot dataSnapshot : snapshot.getChildren()) {
                Password passwords = dataSnapshot.getValue(Password.class);
                passwordslist.add(passwords);
            }
            Collections.sort(passwordslist, Password.sortDescending);
            passwordsAll = new ArrayList<>(passwordslist);
            mAdapter.notifyDataSetChanged();
            gettingpass.setVisibility(View.GONE);
            progressBar.setVisibility(View.GONE);
            if (passwordslist.isEmpty()) {
                nothinghere.setVisibility(View.VISIBLE);
            } else {
                nothinghere.setVisibility(View.GONE);
            }
        }
```

The code snippet above shows that the method will send a request to Firebase to iterate through the given path, which is the child path “passwords” of the current user that contains all the user entries. The *ArrayList* needs to be cleared first before iterating through the database to prevent duplicate entries from showing in the recycler view later on. After adding all the entries into the *ArrayList*, the entries will later be sorted using the *Collections* utility provided by Java based on the item name. After the sorting has been done, it will then make a new copy of the list (shown in the code by setting a new *ArrayList* object of *passwordlist* into a variable called *passwordAll*). This list will be useful later when the user is attempting a search. After that, it will call

Adapter.notifySetDataChanged() to notify the adapter that the content of the list has been changed. This method is called to refresh the recycler view so that it shows the correct items. If the list is empty, it will show a message to the user that they have not been added any entries yet (shown in the code snippet by the name of *nothinghere*).

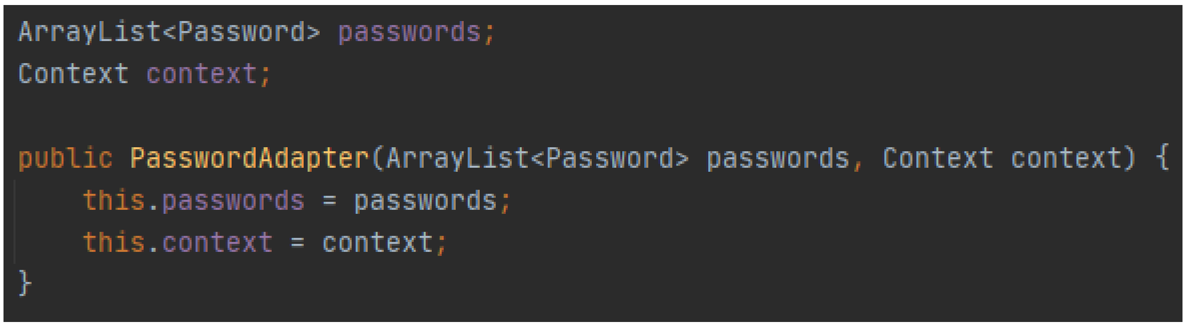


```java
@Override
public Filter getFilter() { return filter; }

Filter filter = new Filter() {
    @Override
    protected FilterResults performFiltering(CharSequence constraint) {
        ArrayList<Password> filteredPasswords = new ArrayList<>();

        if (constraint == null || constraint.length() == 0) {
            filteredPasswords.addAll(passwordsAll);
        } else {
            String filtertext = constraint.toString().toLowerCase().trim();
            for (Password pass : passwordsAll) {
                if (pass.getItemName().toLowerCase().contains(filtertext) ||
                        pass.getUserName().toLowerCase().contains(filtertext)) {
                    filteredPasswords.add(pass);
                }
            }
        }

        FilterResults filterResults = new FilterResults();
        filterResults.values = filteredPasswords;
        return filterResults;
    }

    @Override
    protected void publishResults(CharSequence constraint, FilterResults results) {
        passwordslist.clear();
        passwordslist.addAll((ArrayList<Password>) results.values);
        mAdapter.notifyDataSetChanged();
    }
};
```

The code snippet above is where the search function gets executed. When the user is attempting a search on a specific entry, first of all, it initializes a new temporary list (called *filteredPassword* in the code snippet above). If the user does not input anything into the search bar, it will add all of the entries into the temporary list by making a copy of *passwordAll*, the list that contains all of the entries. Else if the search bar is not empty, filter the entries by the item name and the username and add them into the temporary array. After the filtering has been done, it will call the overridden interface *publishResult()* which will show the filtered entries in the recycler view.

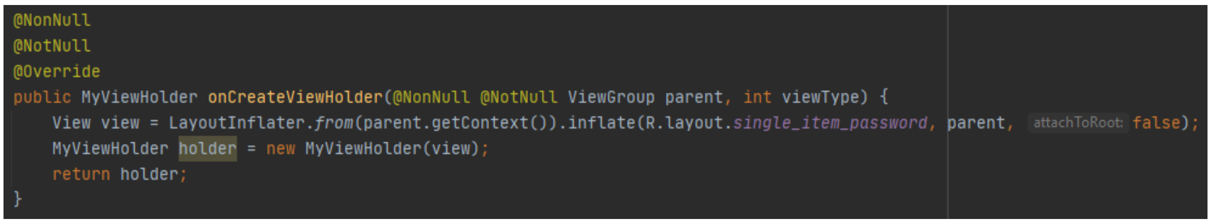
- **Adapter package**

  The class contained in this package acts as an adapter for the *PasswordList*'s *RecyclerView* class. *RecyclerView* is one of the core elements in Android app development. It is widely used as a way to display selectable items in a form of a dropdown list. The simplest example of *RecyclerView* is the "Settings" app in the Android operating system.



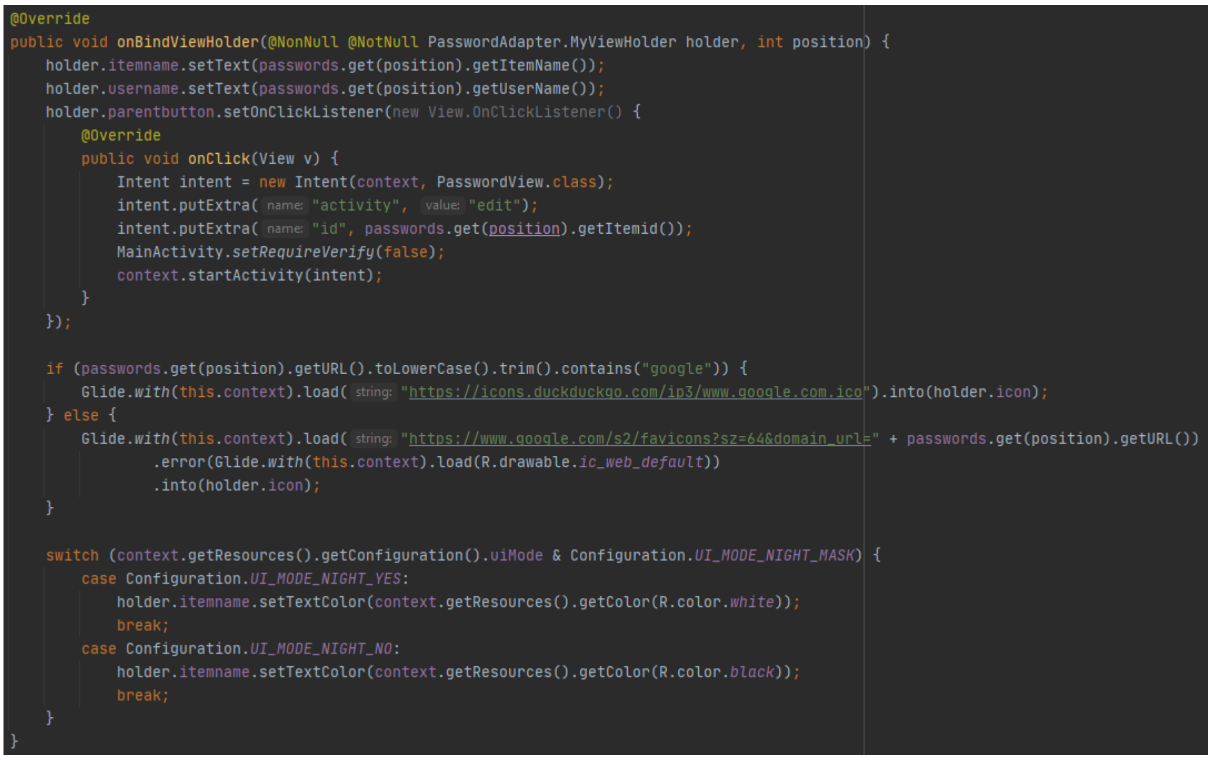
```
ArrayList<Password> passwords;
Context context;

public PasswordAdapter(ArrayList<Password> passwords, Context context) {
    this.passwords = passwords;
    this.context = context;
}
```

The above code snippet is the constructor for the adapter class. It accepts an *ArrayList* and a *Context* as the parameters. The *ArrayList* will be used as the information about the items that are going to be displayed in the recycler view, and the *Context* will be used as the context for the recycler view, since the adapter cannot directly use the *getContext()* method as it is not inherited from *Fragment* nor *AppCompatActivity* class which contains the current activity's context.



```
@NonNull
@NotNull
@Override
public MyViewHolder onCreateViewHolder(@NonNull @NotNull ViewGroup parent, int viewType) {
    View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.single_item_password, parent, attachToRoot: false);
    MyViewHolder holder = new MyViewHolder(view);
    return holder;
}
```

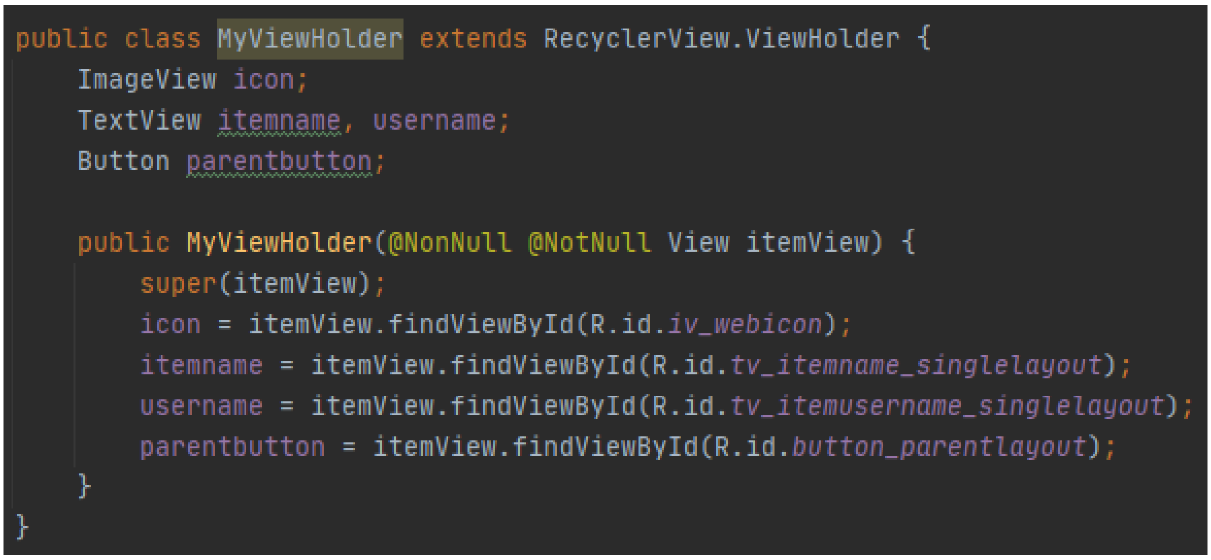
This is the equivalent of the *onCreate()* method in an activity. It acts as the second "constructor" for the adapter class.



```java
@Override
public void onBindViewHolder(@NonNull @NotNull PasswordAdapter.MyViewHolder holder, int position) {
    holder.itemname.setText(passwords.get(position).getItemName());
    holder.username.setText(passwords.get(position).getUserName());
    holder.parentbutton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent intent = new Intent(context, PasswordView.class);
            intent.putExtra( name: "activity",  value: "edit");
            intent.putExtra( name: "id", passwords.get(position).getItemid());
            MainActivity.setRequireVerify(false);
            context.startActivity(intent);
        }
    });

    if (passwords.get(position).getURL().toLowerCase().trim().contains("google")) {
        Glide.with(this.context).load( string: "https://icons.duckduckgo.com/ip3/www.google.com.ico").into(holder.icon);
    } else {
        Glide.with(this.context).load( string: "https://www.google.com/s2/favicons?sz=64&domain_url=" + passwords.get(position).getURL())
                .error(Glide.with(this.context).load(R.drawable.ic_web_default))
                .into(holder.icon);
    }

    switch (context.getResources().getConfiguration().uiMode & Configuration.UI_MODE_NIGHT_MASK) {
        case Configuration.UI_MODE_NIGHT_YES:
            holder.itemname.setTextColor(context.getResources().getColor(R.color.white));
            break;
        case Configuration.UI_MODE_NIGHT_NO:
            holder.itemname.setTextColor(context.getResources().getColor(R.color.black));
            break;
    }
}
```

This is the most important part of the adapter class. Here, the contents of the *ArrayList* gets added into the recycler view, and it will be displayed to the user as the list like the one that is present in the settings app in an Android phone. When an item is clicked, it will call the *onClick()* interface above to open the activity *Password View* to view the entry that the user has selected from the recycler view.

The class *PasswordAdapter* class contains an additional view holder class which contains information about the elements in the recycler view object.



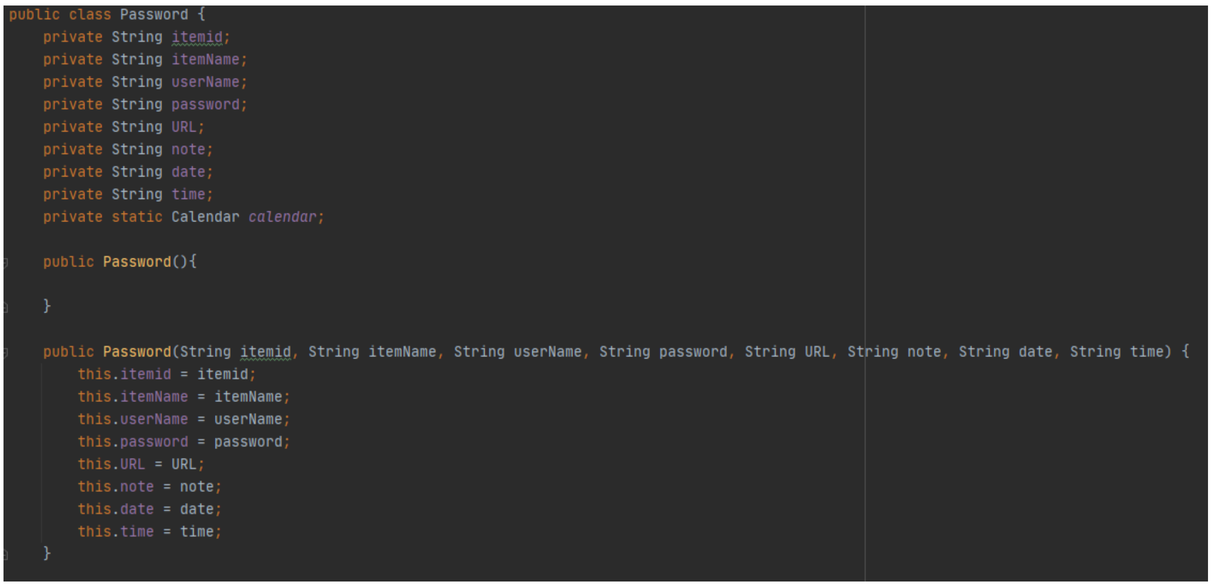
```java
public class MyViewHolder extends RecyclerView.ViewHolder {
    ImageView icon;
    TextView itemname, username;
    Button parentbutton;

    public MyViewHolder(@NonNull @NotNull View itemView) {
        super(itemView);
        icon = itemView.findViewById(R.id.iv_webicon);
        itemname = itemView.findViewById(R.id.tv_itemname_singlelayout);
        username = itemView.findViewById(R.id.tv_itemusername_singlelayout);
        parentbutton = itemView.findViewById(R.id.button_parentlayout);
    }
}
```

MyViewHolder class contains information about the items inside a single recycler view item. In Password Note's version of a single recycler view item, it contains an icon (*icon)*, two texts (*itemname* and *username)*, and an invisible button (*parentbutton)* to be used as a sign that the item is clickable.

- **Model package**

  The model package is where the base of the operation of this app is. There is only one class in this package, the "Password". As the name states, "Password" contains the instances of a user entry.

The Password Class



```java
public class Password {
    private String itemid;
    private String itemName;
    private String userName;
    private String password;
    private String URL;
    private String note;
    private String date;
    private String time;
    private static Calendar calendar;

    public Password(){

    }

    public Password(String itemid, String itemName, String userName, String password, String URL, String note, String date, String time) {
        this.itemid = itemid;
        this.itemName = itemName;
        this.userName = userName;
        this.password = password;
        this.URL = URL;
        this.note = note;
        this.date = date;
        this.time = time;
    }
```

As stated before, this class contains the instances of an entry object. This means that this class contains mostly of getters and setters for the instances. The code snippet above is the constructor for the *Password* class.

- *itemid*

  This instance will hold the ID that will be generated automatically by Firebase later on when the entry has been added into the database.

- *itemName*

  This instance will hold the item name that the user will input later on when making a new entry.

- *userName*

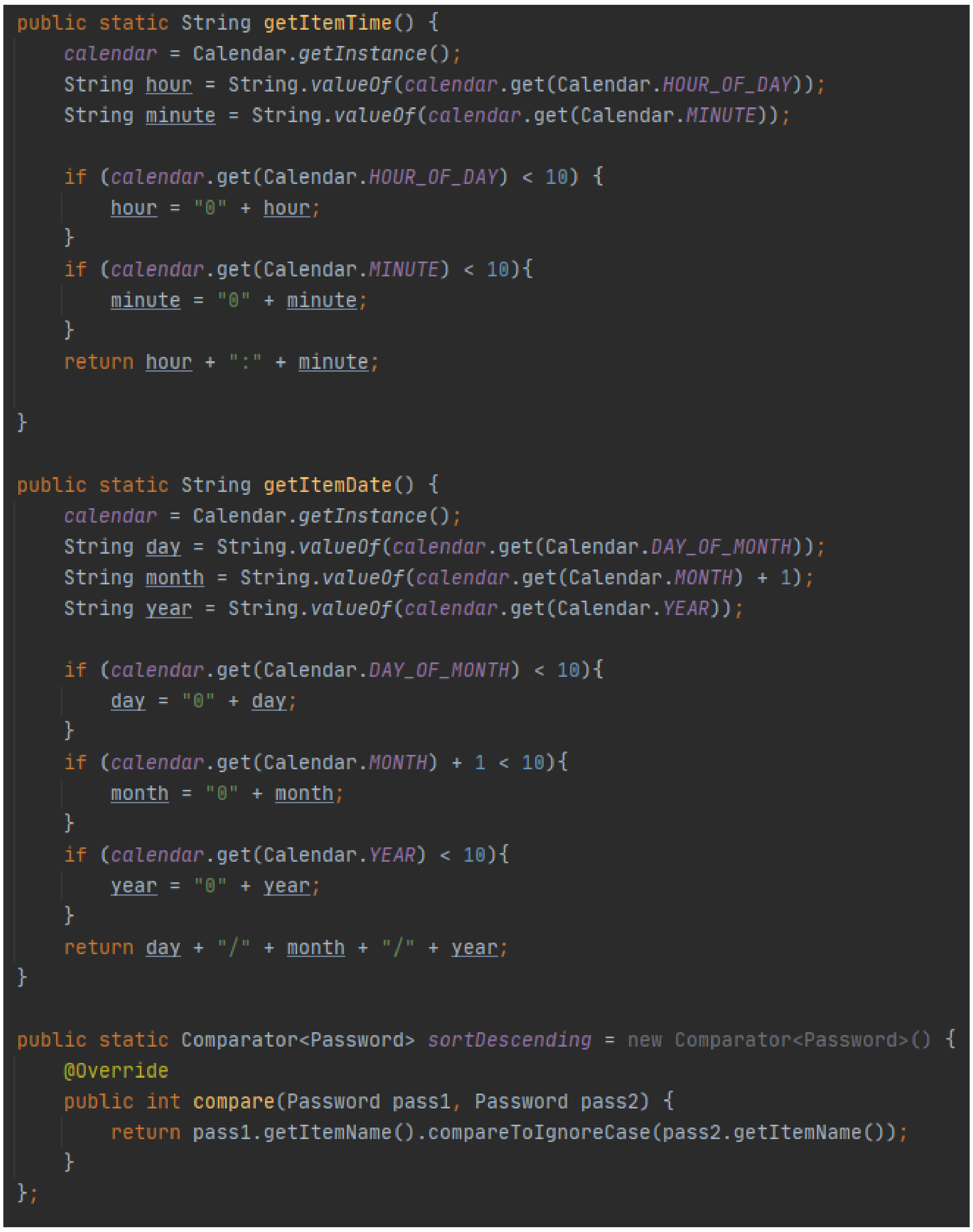
  As the instance name states, this instance will hold the username for the entry that the user will input later on.
- *password*

  This instance will hold the password information that the user will input later when making a new entry.
- *URL*

  This instance will hold the website URL for the user entry. This instance will also be used to set the items icon in the recycler view.
- *note*

  This instance will hold the entry note if the user provides any.
- *date* and *time*

  Both of these instances will hold when the entry was made or modified by the user. The instance *date* will hold the date creation or modification of the entry, and the instance *time* will hold the time of the creation or modification. The time will be shown in 24-hour format.


```java
public static String getItemTime() {
    calendar = Calendar.getInstance();
    String hour = String.valueOf(calendar.get(Calendar.HOUR_OF_DAY));
    String minute = String.valueOf(calendar.get(Calendar.MINUTE));

    if (calendar.get(Calendar.HOUR_OF_DAY) < 10) {
        hour = "0" + hour;
    }
    if (calendar.get(Calendar.MINUTE) < 10){
        minute = "0" + minute;
    }
    return hour + ":" + minute;

}

public static String getItemDate() {
    calendar = Calendar.getInstance();
    String day = String.valueOf(calendar.get(Calendar.DAY_OF_MONTH));
    String month = String.valueOf(calendar.get(Calendar.MONTH) + 1);
    String year = String.valueOf(calendar.get(Calendar.YEAR));

    if (calendar.get(Calendar.DAY_OF_MONTH) < 10){
        day = "0" + day;
    }
    if (calendar.get(Calendar.MONTH) + 1 < 10){
        month = "0" + month;
    }
    if (calendar.get(Calendar.YEAR) < 10){
        year = "0" + year;
    }
    return day + "/" + month + "/" + year;
}

public static Comparator<Password> sortDescending = new Comparator<Password>() {
    @Override
    public int compare(Password pass1, Password pass2) {
        return pass1.getItemName().compareToIgnoreCase(pass2.getItemName());
    }
};
```

Aside from the getters and setters, the *Password* class also contains three additional methods:

- *getItemTime()*

  This method will be used to get the current time using the *Calendar* class. This will be used to set the modification time of an entry.

- *getItemDate()*

  This method is basically the same as *getItemTime()*, but is used to get the current date in a form of dd/mm/yyyy to be later used to set the modification date of an entry.

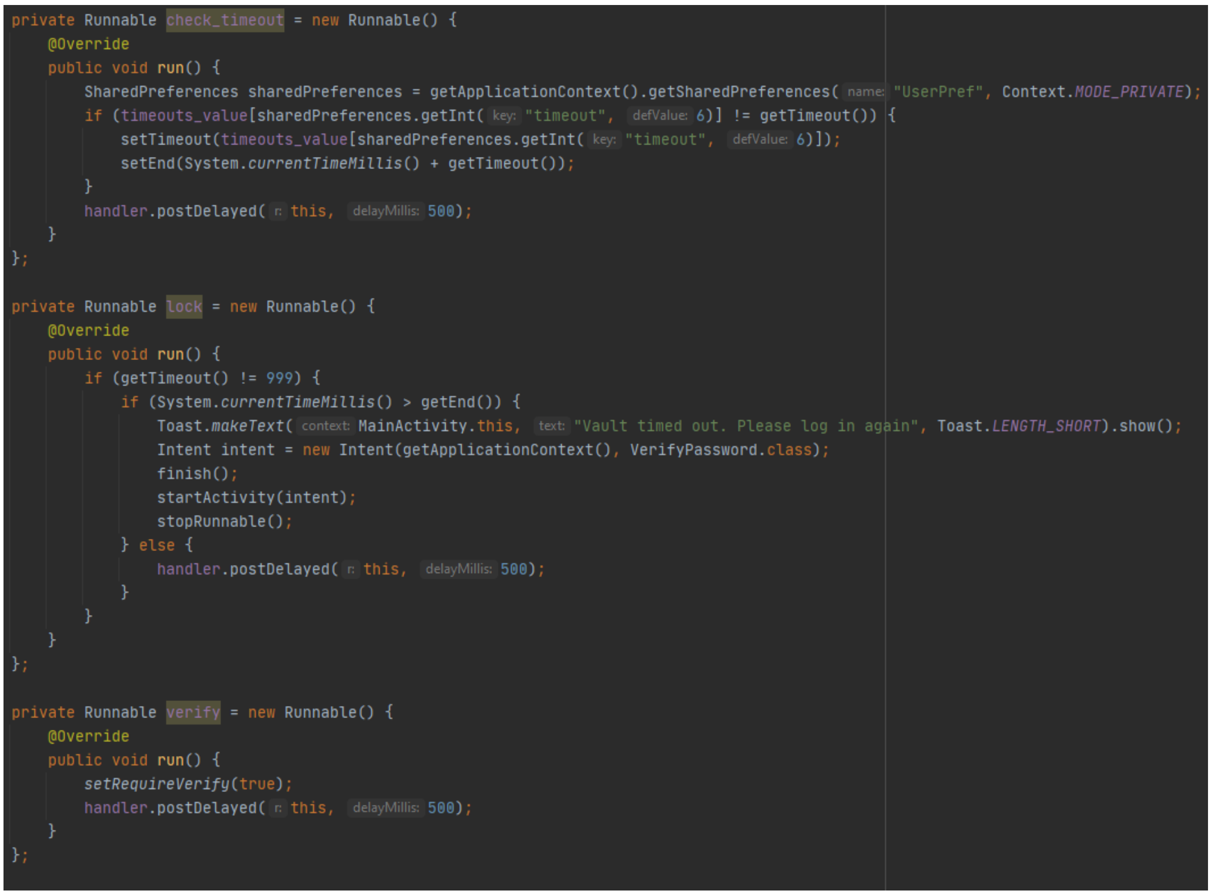
- *sortDescending()*

  This method is used correspondingly with the *Collection* class in *Password List* activity to sort the retrieved items from the database in descending order based on the entries' item names.

- **Miscellaneous package**

  This contains three classes, they are *MainActivity*, *SplashScreen*, and *SecureView*. *MainActivity* is the main activity of this application. It contains the three fragments that has been explained above. *SplashScreen* is there to check whether the app already has a signed in user or not when the app is first launched (not re-opened from background applications). If there is already a signed in user, then it will tell the application to go to the *Verify Password* activity, else, it will tell the application to go to the *Login* activity. *SecureView* is a class which defines the instruction for the application to prevent the user from screenshotting the application.

a. Main Activity Class



```
private Runnable check_timeout = new Runnable() {
    @Override
    public void run() {
        SharedPreferences sharedPreferences = getApplicationContext().getSharedPreferences( name: "UserPref", Context.MODE_PRIVATE);
        if (timeouts_value[sharedPreferences.getInt( key: "timeout",  defValue: 6)] != getTimeout()) {
            setTimeout(timeouts_value[sharedPreferences.getInt( key: "timeout",  defValue: 6)]);
            setEnd(System.currentTimeMillis() + getTimeout());
        }
        handler.postDelayed( r: this,  delayMillis: 500);
    }
};

private Runnable lock = new Runnable() {
    @Override
    public void run() {
        if (getTimeout() != 999) {
            if (System.currentTimeMillis() > getEnd()) {
                Toast.makeText( context: MainActivity.this,  text: "Vault timed out. Please log in again", Toast.LENGTH_SHORT).show();
                Intent intent = new Intent(getApplicationContext(), VerifyPassword.class);
                finish();
                startActivity(intent);
                stopRunnable();
            } else {
                handler.postDelayed( r: this,  delayMillis: 500);
            }
        }
    }
};

private Runnable verify = new Runnable() {
    @Override
    public void run() {
        setRequireVerify(true);
        handler.postDelayed( r: this,  delayMillis: 500);
    }
};
```

There are three runnables in this class:

- *check_timeout*

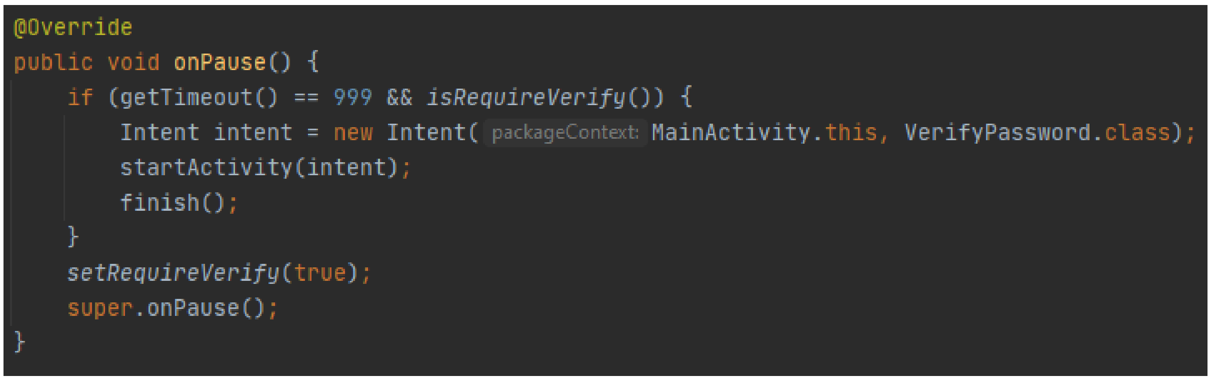
  This runnable will always be run on the background once every 500 milliseconds. This is done to check the user timeout configuration change. Apparently, I found no way for getting a value from a dialogue box. That is why I used the *SharedPreferences* class to save the user configuration in a form of a key-value pair. Other than to save the user configuration, it is actually used to get the configuration value that is used in this runnable too.

- *lock*

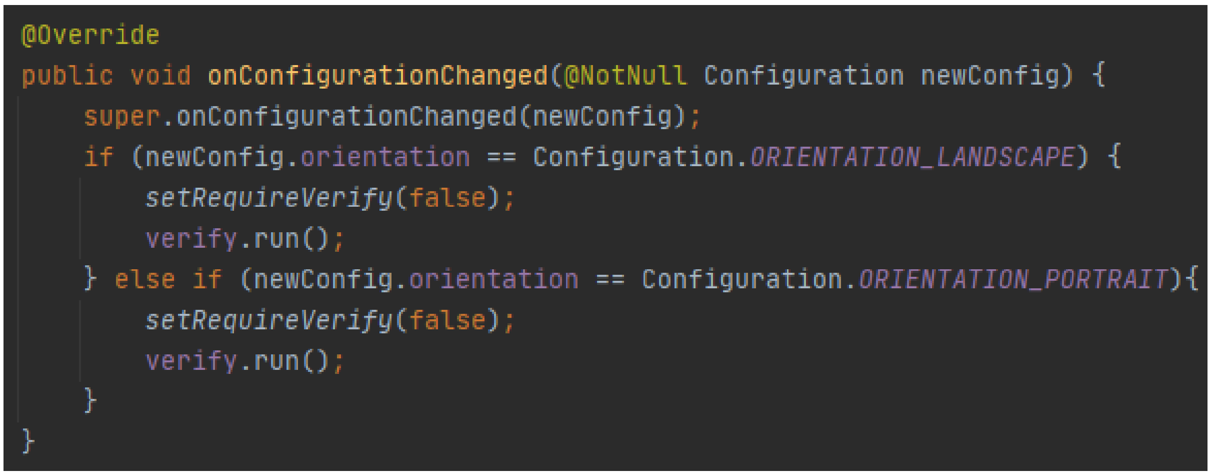
  This runnable will also be run on the background alongside *check_timeout.* This runnable is made to check the timeout if the user does not choose "immediate lock after exit" as the configuration for the vault timeout.

- *verify*

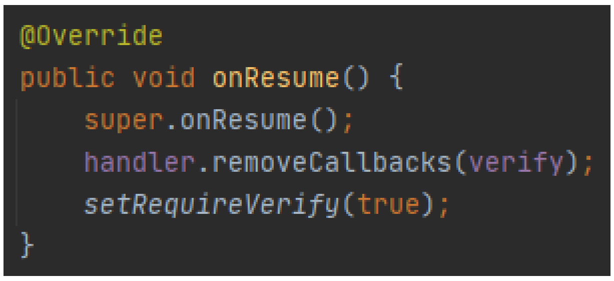
  As explained before, whenever the orientation changes, the current activity will go on a paused state and it will call the *onPause()* method. After the orientation has changed successfully, it will then call the *onResume()* method. If this runnable does not exist, whenever the orientation changes, the user has to verify their password. This is very inconvenient as the phone can regularly switch orientations depending on the user's need. This runnable is the solution for that. As for why the user has to verify their password whenever the activity goes on a paused state, it will later be explained in its own section.


```
@Override
public void onPause() {
    if (getTimeout() == 999 && isRequireVerify()) {
        Intent intent = new Intent( packageContext: MainActivity.this, VerifyPassword.class);
        startActivity(intent);
        finish();
    }
    setRequireVerify(true);
    super.onPause();
}
```

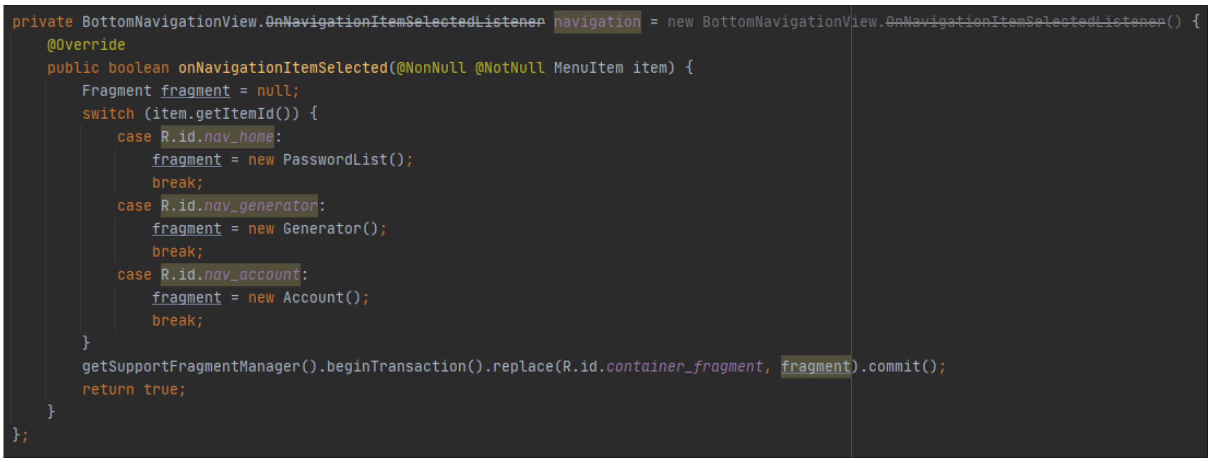
onPause() and *onResume()* are overridden in this activity. Continuing the explanation from before, the user has to verify their password whenever the activity goes on paused state is because when the user leaves the application and they enter it again, if they choose "immediate lock after exit" configuration, logically, they will need to verify their password before they can see their vault. This is why I implemented that functionality in the *onPause()* method. However, because the activity gets paused too whenever the orientation changes, I have to find a solution to prevent the app from locking the user out. The solution is the *verify* runnable that has been explained before.


```
@Override
public void onConfigurationChanged(@NotNull Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
        setRequireVerify(false);
        verify.run();
    } else if (newConfig.orientation == Configuration.ORIENTATION_PORTRAIT){
        setRequireVerify(false);
        verify.run();
    }
}
```

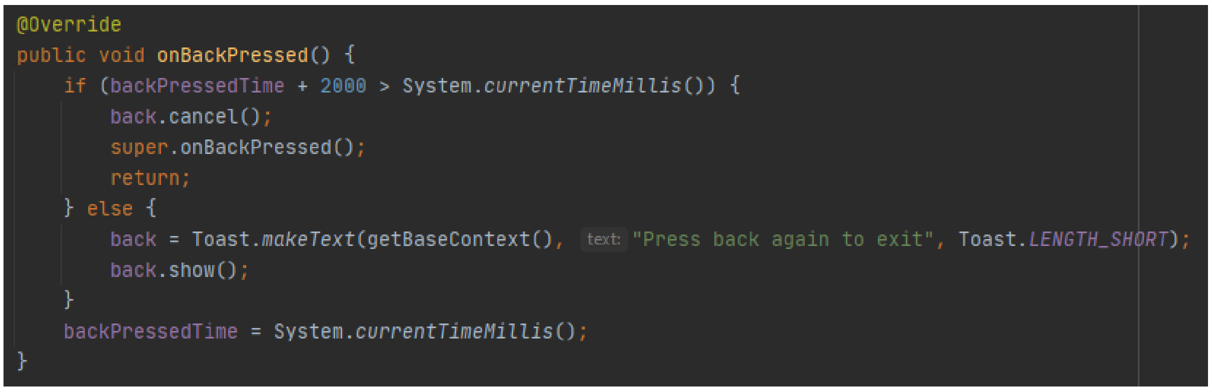
Whenever the orientation changes, *setRequireVerify()* becomes *false*. That way, it tells the *onPause()* method that "You don't need to lock the user out, they are only changing the orientation of the device".


```java
@Override
public void onResume() {
    super.onResume();
    handler.removeCallbacks(verify);
    setRequireVerify(true);
}
```

After the activity got resumed, however, the runnable will be removed and the *setRequireVerify()* will be back to true, meaning that it will tell the *onPause(),* “There, the orientation have changed. I have set the *setRequireVerify* back to *true,* so you will need to lock the user out if they go out from the application”.

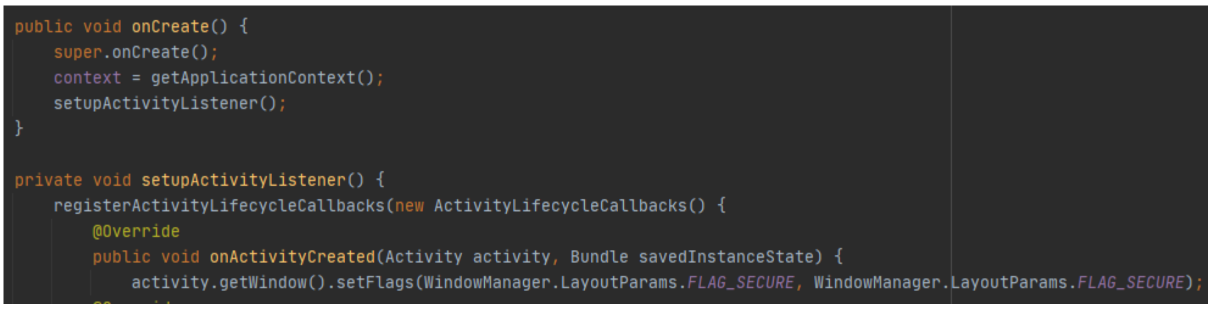

```java
private BottomNavigationView.~~OnNavigationItemSelectedListener~~ navigation = new BottomNavigationView.~~OnNavigationItemSelectedListener~~() {
    @Override
    public boolean onNavigationItemSelected(@NonNull @NotNull MenuItem item) {
        Fragment fragment = null;
        switch (item.getItemId()) {
            case R.id.nav_home:
                fragment = new PasswordList();
                break;
            case R.id.nav_generator:
                fragment = new Generator();
                break;
            case R.id.nav_account:
                fragment = new Account();
                break;
        }
        getSupportFragmentManager().beginTransaction().replace(R.id.container_fragment, fragment).commit();
        return true;
    }
};
```

This is the bottom navigation bar programming. And lastly,


```java
@Override
public void onBackPressed() {
    if (backPressedTime + 2000 > System.currentTimeMillis()) {
        back.cancel();
        super.onBackPressed();
        return;
    } else {
        back = Toast.makeText(getBaseContext(), text: "Press back again to exit", Toast.LENGTH_SHORT);
        back.show();
    }
    backPressedTime = System.currentTimeMillis();
}
```

This is the overridden method *onBackPressed()* which tells the application that if the user pressed the back button for the first time, show the message and set the *backPressedTime* to current system time (in milliseconds). If until 2 seconds later the user does not press the back button again, that means that the user has decided not to exit from the application yet, so whenever they press the back button again, show the message again. If they press the button during the 2 seconds timer, then the application will know that the user is intended to leave the application.

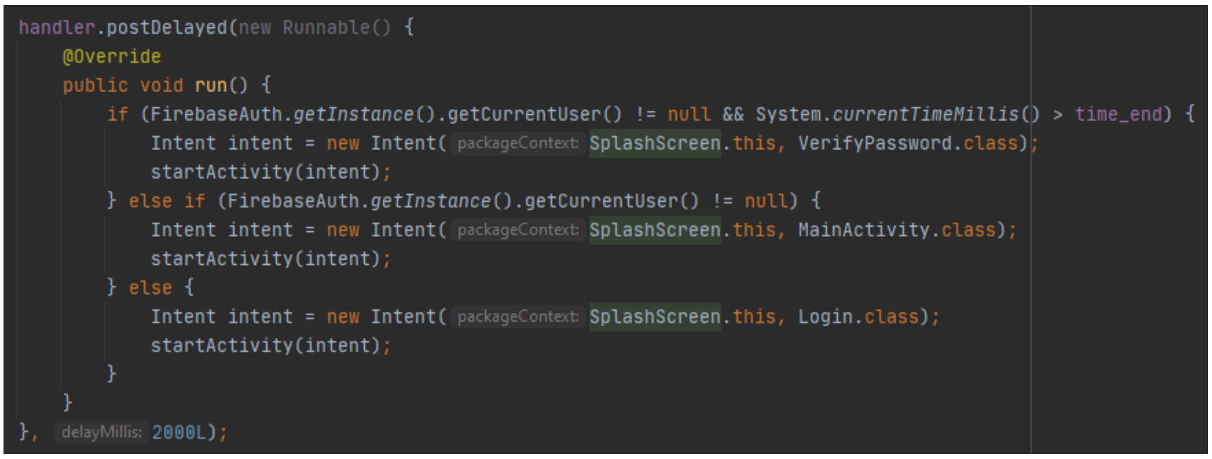
b. Secure View Class



```java
public void onCreate() {
    super.onCreate();
    context = getApplicationContext();
    setupActivityListener();
}

private void setupActivityListener() {
    registerActivityLifecycleCallbacks(new ActivityLifecycleCallbacks() {
        @Override
        public void onActivityCreated(Activity activity, Bundle savedInstanceState) {
            activity.getWindow().setFlags(WindowManager.LayoutParams.FLAG_SECURE, WindowManager.LayoutParams.FLAG_SECURE);
```

As briefly explained before, this class contains the instruction for the application to prevent the user from screen capturing the application in any activity that they are currently in.

c. Splash Screen Class



```java
handler.postDelayed(new Runnable() {
    @Override
    public void run() {
        if (FirebaseAuth.getInstance().getCurrentUser() != null && System.currentTimeMillis() > time_end) {
            Intent intent = new Intent( packageContext: SplashScreen.this, VerifyPassword.class);
            startActivity(intent);
        } else if (FirebaseAuth.getInstance().getCurrentUser() != null) {
            Intent intent = new Intent( packageContext: SplashScreen.this, MainActivity.class);
            startActivity(intent);
        } else {
            Intent intent = new Intent( packageContext: SplashScreen.this, Login.class);
            startActivity(intent);
        }
    }
},  delayMillis: 2000L);
```

Runnable.postDelayed() means that the instructions contained in the *postDelayed()* will be run after a set amount of time. In this case, it will run after 2 seconds. During the 2 seconds, the activity will show Password Note's logo. After 2 seconds have passed, if there is a user that was logged in to the app on the previous exit, then start a new *Verify Password* activity, else, start a new *Login* activity.

IX. Project and Video Link

1. Project Repository and demo video
   https://github.com/karelbondan/password_notes
2. HD UML diagrams
   https://drive.google.com/drive/folders/1l9bXrd7Mn_IKq3ti7YJ0_0YGdcBCQ_B4?usp=sharing

X. References

https://www.youtube.com/playlist?list=PLywoMLAauVYD1h8t72sLxuHIxCc_XyvRl