ANDROID APP DEVELOPMENT IN JAVA

Structure of Computer Systems project

JANUARY 1, 2024

COT MIHAITA MARIAN
Technical University of Cluj-Napoca

Contents

Introduction

1.1 Overview

Welcome to the Mobile App Development Tutorial, where innovation meets hands-on learning! In this dynamic tutorial series, we delve into the fascinating world of Java-based Android app development, equipping you with the skills to create three unique and impactful mobile applications.

In today's digital era, mobile applications have become integral to our daily lives, offering solutions to diverse needs, and enhancing user experiences. This tutorial is designed to provide you with a comprehensive understanding of mobile app development, from concept to execution.

1.2 Objectives

This tutorial is designed to provide a well thought roadmap for efficient learning of mobile development. This plan consists of development of three android mobile applications scaling in difficulty from easy to medium and in the end to a hard one.

This project will offer a wide variety of information and concepts which will help setting a good base for any student trying to develop mobile engineering skills.

Theoretical Concepts

2.1 Prerequisites

Before diving into the project, ensure that you have the following prerequisites in place:

1. Java Proficiency:
    - A solid understanding of core Java concepts is essential for developing Android applications. If you are not familiar with Java, we recommend brushing up on basic syntax, object-oriented programming principles, and common data structures.

2. Android Studio Installation:

    - The project is developed using Android Studio, the official integrated development environment (IDE) for Android app development. Make sure you have Android Studio installed on your machine. You can download it from the official Android Studio website.
    - If you are new to Android Studio, consider going through the official Getting Started Guide provided by Google.

2.2 Tech Stack

Programming Languages:

1. **Java:**
    - The primary programming language for developing the Android applications. Java is used for writing the business logic, implementing the application's functionality, and managing data.
2. **XML (eXtensible Markup Language):**
    - XML is utilized for defining the structure and layout of user interface elements in Android. Layouts and UI elements are described using XML to achieve a clear separation between the application's presentation and business logic.

Development Framework:

1. Android SDK (Software Development Kit):
    - The Android SDK provides a comprehensive set of tools and libraries for Android app development. It includes APIs for interacting with the Android system, managing user interfaces, and handling various functionalities such as database operations and network communication.

User Interface Design:

1. XML Layouts:
    - XML is extensively used to define the layout and appearance of user interfaces. Android's XML-based layout files allow for a declarative approach to designing UI elements, making it easier to visualize and organize the structure of the app's screens.

2. Material Design Components:
    - Leveraging Android's Material Design components to create a consistent and visually appealing user interface. Material Design principles guide the design of UI elements, ensuring a modern and intuitive user experience.

Easy Application: Input Alert

3.1 Overview

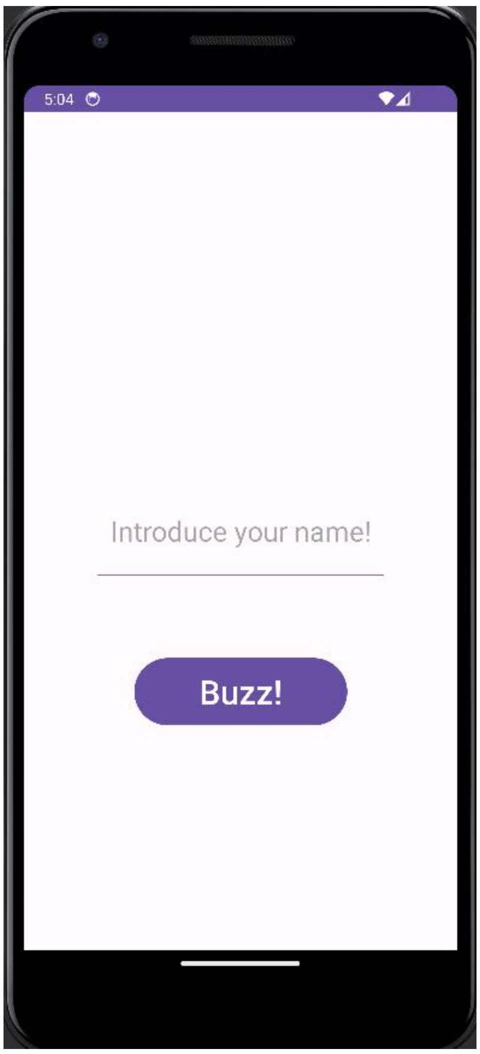
To start of this tutorial at a good pace we'll be tackling firstly an easy application to implement called Input Alert. In essence this application consists of a labelled field where the user can input anything related to a message and a "Buzz" button which will then alert the user the message which was written in the label.

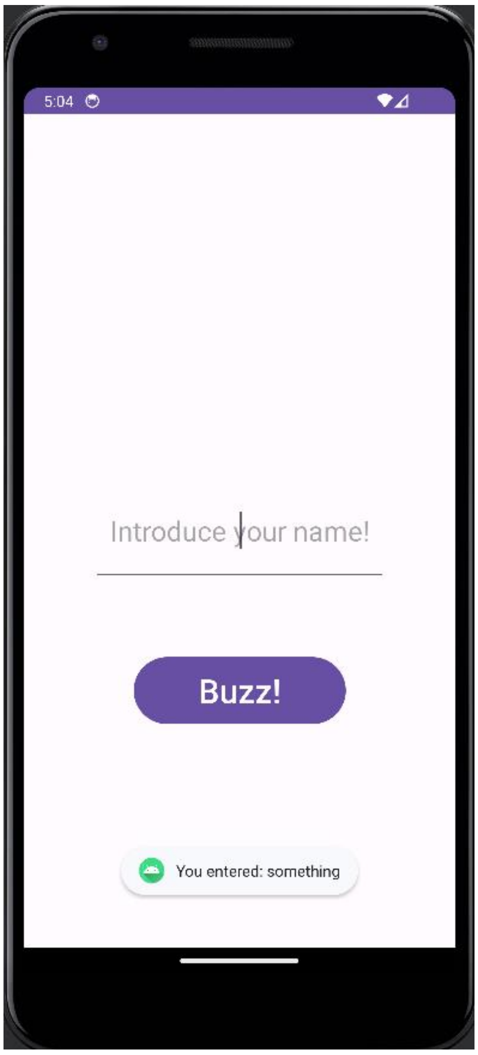
3.2 Key features

This project is extremely easy having the purpose of getting you accustomed with Java and Android Studio altogether.

As one can see in the images displayed bellow, the app has a simple layout consisting of the start up activity, a text field, and a button. For convenience purposes the text field will be emptied after the button was pressed to allow the easier usage of the functionality.

To implement alerts, we don't have to do anything too complicated since they are a basic concept and just have to be called upon.

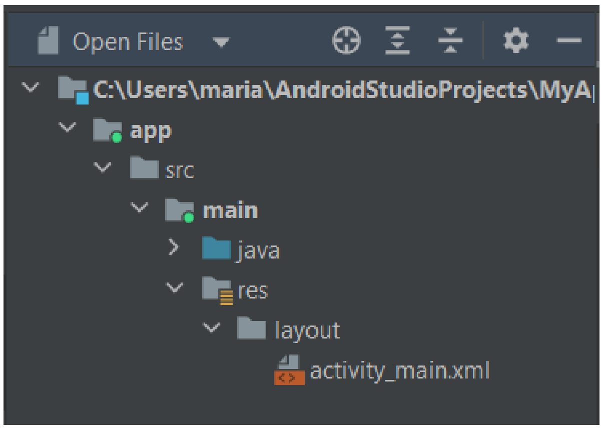




3.3 Development

For the development process we'll start with the layout of the application, taking care at first of the way it looks, elements that need to be placed on the screen and constrains applied for the respective elements.

To start of the development phase, first create an empty activity application then head to the activity_main.xml file. Here will be the place where we can encode how our interface should look like.



The IDE chosen for this project is reliable because it offers ways to visualize your layout in real time and having the option of drag and drop for the elements which should create the layout of the apps.

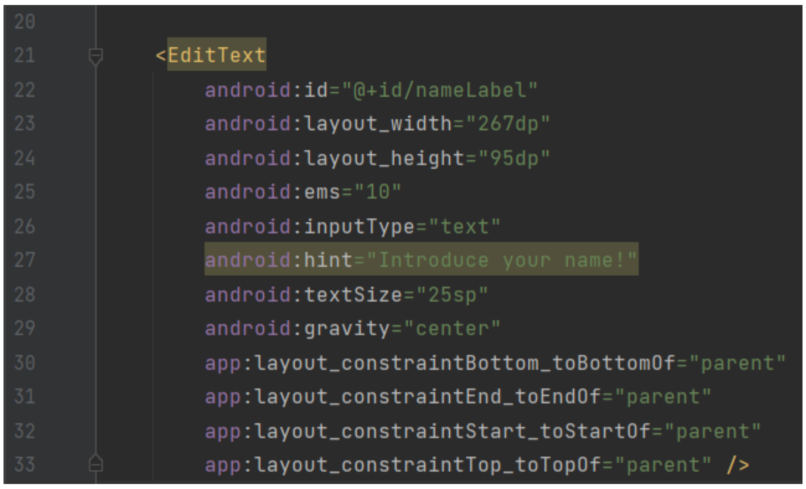
For the current application all we need to set down is a label and a button, so we shall get to work:

As you can see in the images presented below, we'll first be addressing the text component. This component will be a EditText component which is in essence a simple text view which can be edited at will.

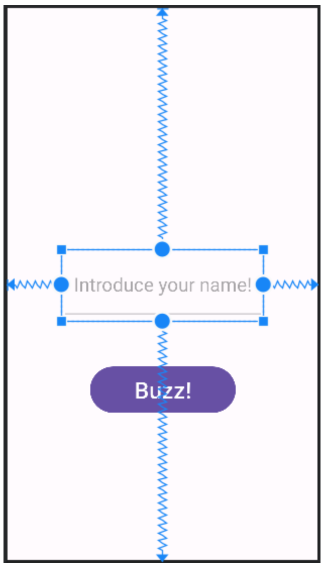
To set down this label properly we'll first set it's constraining, which you can notice at the bottom of the code under the field of app:layout_constrain. To exemplify what this constrains are; they are simply anchors meant to keep your elements in place when the application starts. You have to set down at least two of this constrains to ensure your elements starts at the right position, if this aren't set your element will simply start in the top left corner of the screen and thus if you'd have more elements, they'd overlap each other and become a mess.

Next what we need to address is the id, this is a very important component which can be thought as the actual name of the element itself since by this id we'll be able to call and modify the element inside out Java file. I've decided to name this as "nameLabel" and give it a hint of "Introduce your name!" so the user knows what this field should be used for exactly. For this we'll also give it an input type of text.

The text size and dimensions can be chosen after preference.

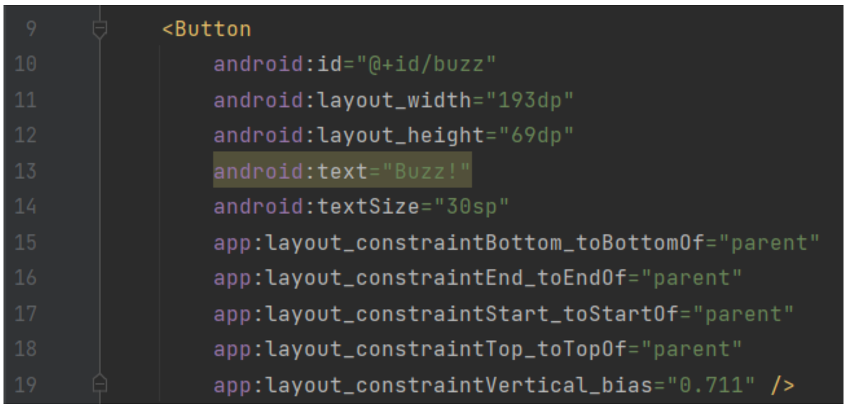


```
    <EditText
        android:id="@+id/nameLabel"
        android:layout_width="267dp"
        android:layout_height="95dp"
        android:ems="10"
        android:inputType="text"
        android:hint="Introduce your name!"
        android:textSize="25sp"
        android:gravity="center"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
```

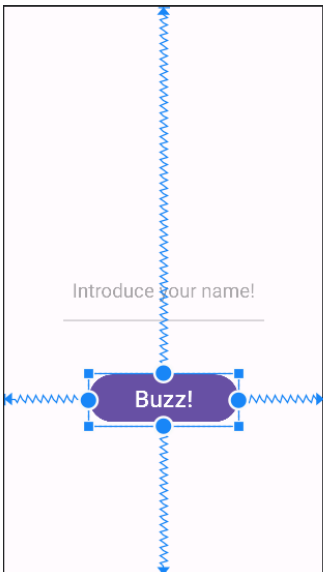


Now that the text field was taken care of, we'll pass to the button.

Just like with the text field, we'll first be setting up our constrains and id, give it a text of "Buzz!" and the dimensions we need to, you can always change its dimensions at will so this is not something standard, only an example.



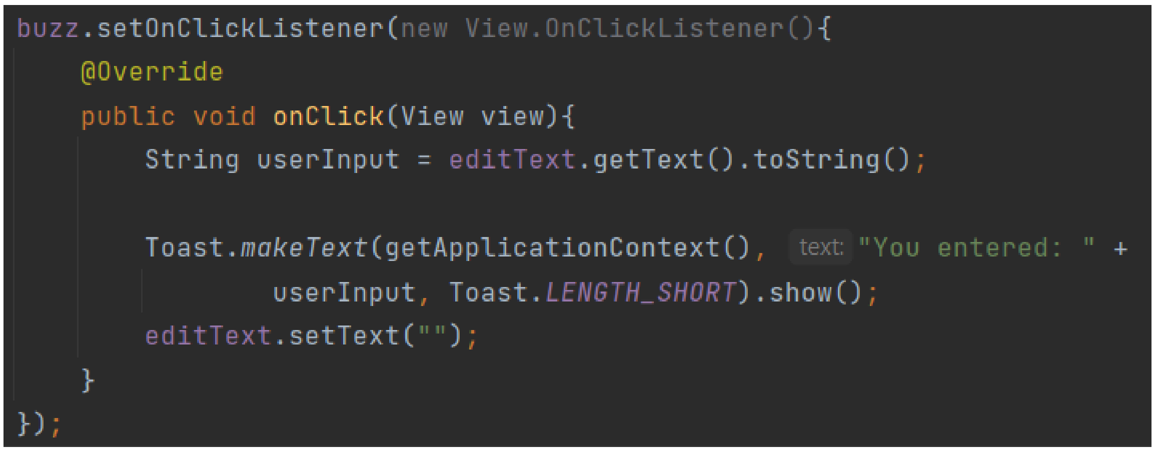
```
<Button
    android:id="@+id/buzz"
    android:layout_width="193dp"
    android:layout_height="69dp"
    android:text="Buzz!"
    android:textSize="30sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.711" />
```



For the actual logic behind this application, we'll have to access our Java file under the name of MainActivity. This file right here contains the logic of the start-up view, since the app is a single view, we don't necessarily need more different files and can work up the logic behind the alert in the start-up one.

For the logic behind we'll start by creating two local variables of type EditText and Button with their names being editText and buzz respectively. In the onCreate method we already have the content set to our current main activity view, so we'll continue by assigning these variables declared to their respective elements by using the method **findViewById**, which will set our variables to their elements via the given ids, why I said on top the ids are very important.

As seen in the image bellow, we create the logic for the click action inside the **setOnClickListener** method for the buzz variable. This method will add the following described actions to the click event of the battle. At first we get the text from the text view, parse it to string format, then create the toast. We give the method the app context, and what to display, in this case the strings presented bellow and the length of the toast. The show method is there to display the toast after creating it. At the end we simply set the text to an empty string to clear the input.



```
buzz.setOnClickListener(new View.OnClickListener(){
    @Override
    public void onClick(View view){
        String userInput = editText.getText().toString();

        Toast.makeText(getApplicationContext(), text: "You entered: " +
                userInput, Toast.LENGTH_SHORT).show();
        editText.setText("");
    }
});
```

This should be it for the first application. Developing this we learned how to handle elements on screen mostly, set their constraints and ids but also how to access them. This are base concepts that will be useful in the later app development of this project.

Medium application: Gun Simulator

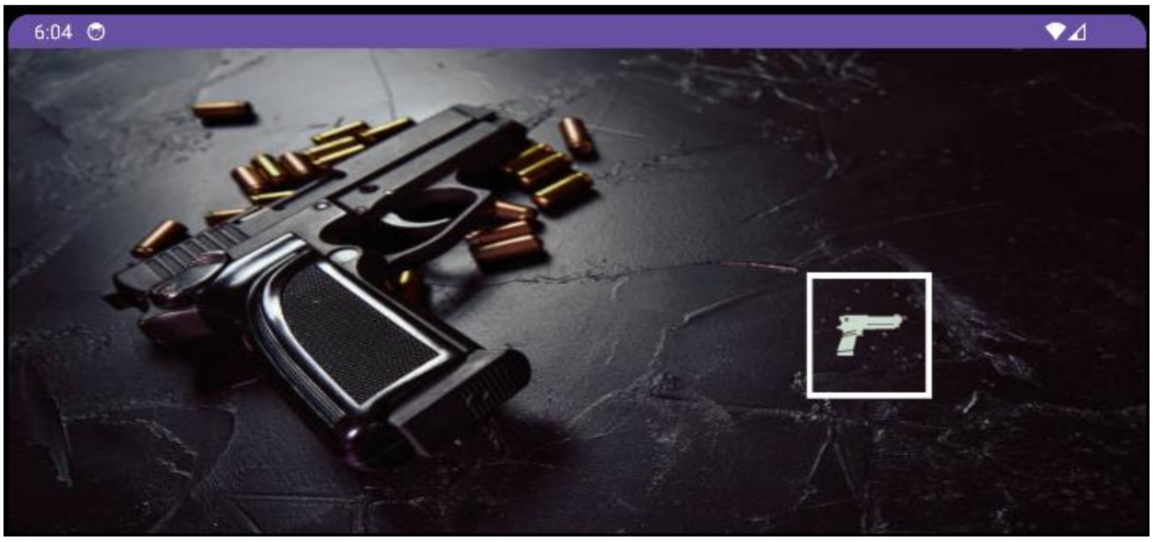
4.1 Overview

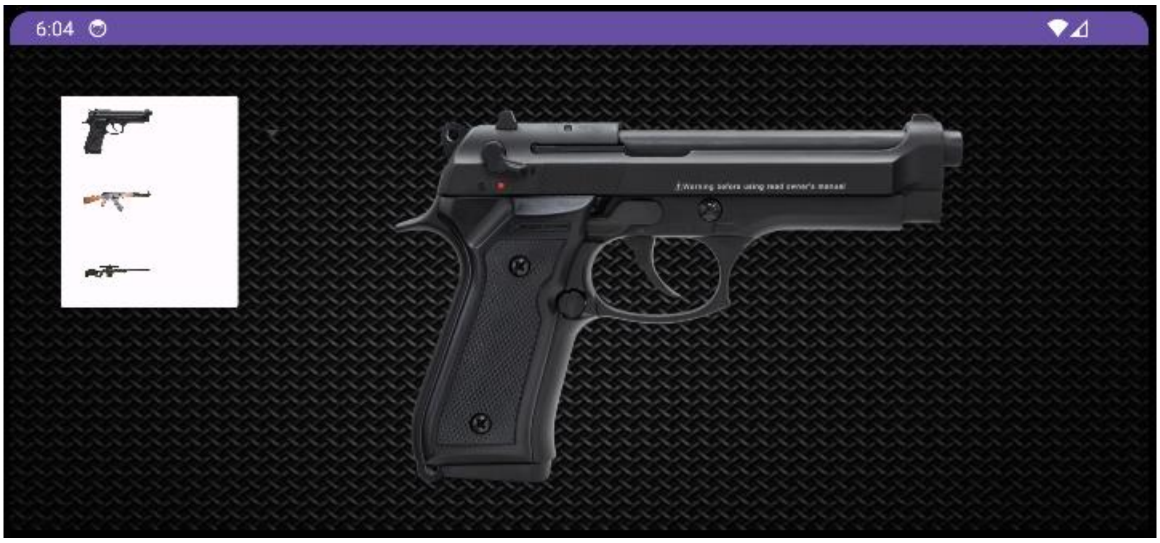
Okey, it is now time to start the development of something more challenging. For this I've chosen to design a gun simulator application which will include multiple views, sound and animation triggers, external elements and work around with drawable items.

4.2 Key Features

The key features present in this app are the fact we'll be using animations and external components. The app will present a starting screen with a button which will start the main application. In the main view we'll have our gun which can shoot, triggering a sound and an animation, to swap guns we'll design a spinner adapter.

The gun will have a button which will trigger the animation and sound of the shot.

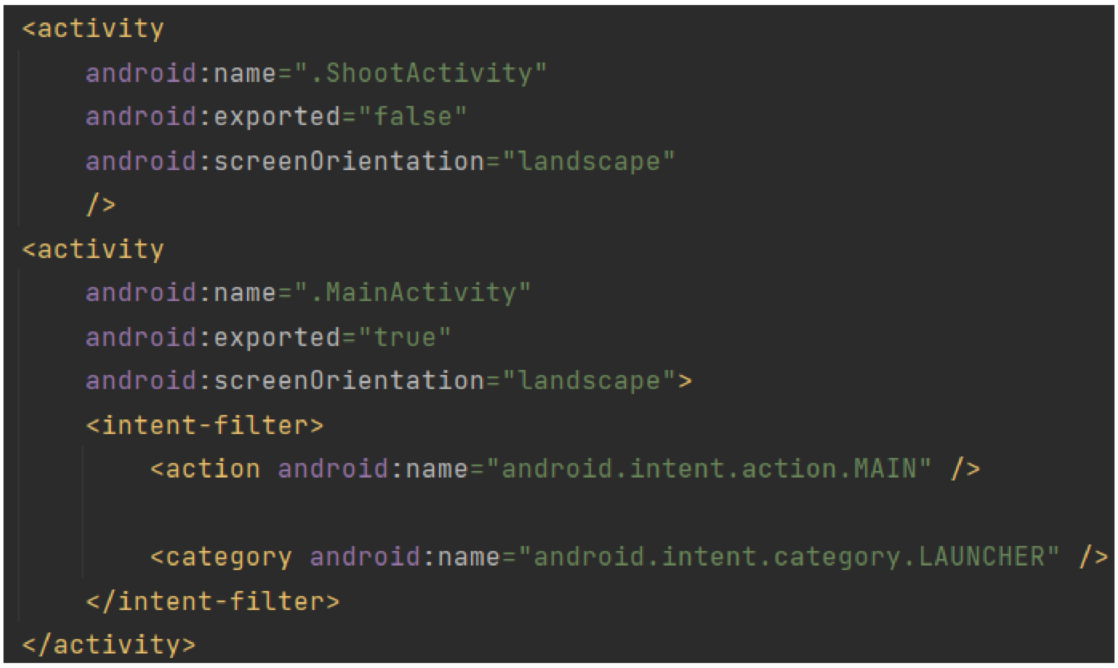




4.3 Development

To start off the development of this second application we'll first need to address the orientation of the layout. I've planned for this app to be used only in landscape mode, so we'll go ahead and impose this constrain on the main activity inside the manifest.xml file:

In the image bellow I've set up the landscape for both the main activity and the shoot activity, the main activity is representing the start up screen while the shoot activity represents the view where the gun and shooting action will be implemented.



```
<activity
    android:name=".ShootActivity"
    android:exported="false"
    android:screenOrientation="landscape"
    />
<activity
    android:name=".MainActivity"
    android:exported="true"
    android:screenOrientation="landscape">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

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

Start-up activity

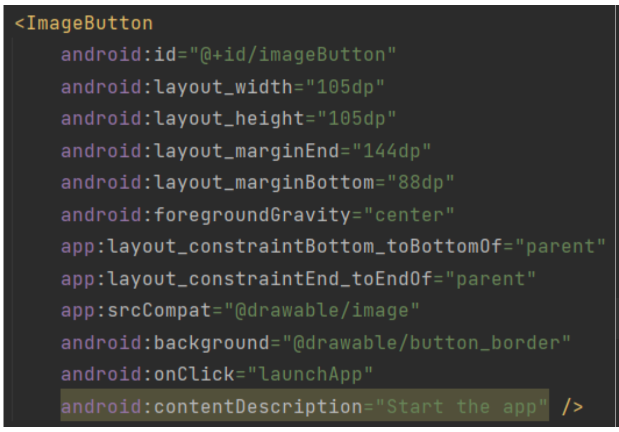
For the start-up activity, we'll start by creating the layout, here we need to set the background of the whole activity to the picture given in the project resources and make use of the drawable folder. In the drawable folder we'll be storing the icons and pictures we need to implement this application. For this action we simply set the background to the id of the image we want "@drawable/overlay" in this case.

Now we need to create the button, for this button we'll be using an image button element which will receive an image as background and a separate view which will act as a border for the button.

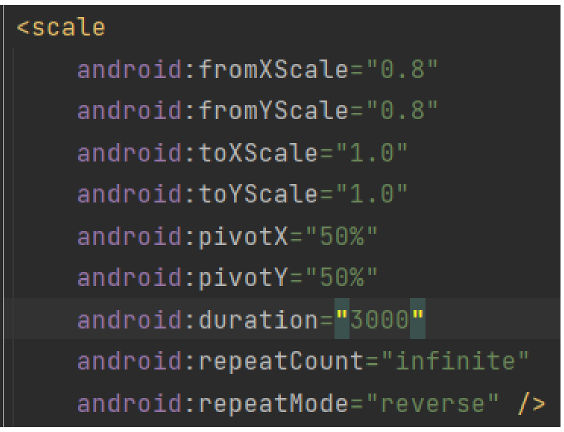
In the drawable folder we'll create a new xml file which will act as a separate item. This item will be furtherly used as a background for our button to easily create a border as shown in the image above.

To complete this button, we need to create a scaling animation, timed so it makes the button grow and shrink creating an effect making the button easily spottable and easier to click.

This animation file will be created in the anim folder as a separate xml file containing the code showed below.



```xml
<ImageButton
    android:id="@+id/imageButton"
    android:layout_width="105dp"
    android:layout_height="105dp"
    android:layout_marginEnd="144dp"
    android:layout_marginBottom="88dp"
    android:foregroundGravity="center"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:srcCompat="@drawable/image"
    android:background="@drawable/button_border"
    android:onClick="launchApp"
    android:contentDescription="Start the app" />
```

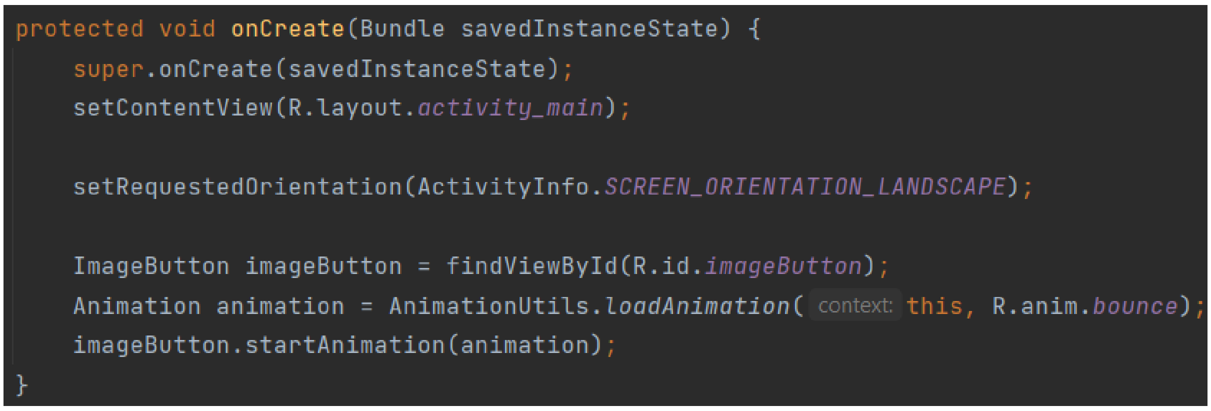


```xml
<scale
    android:fromXScale="0.8"
    android:fromYScale="0.8"
    android:toXScale="1.0"
    android:toYScale="1.0"
    android:pivotX="50%"
    android:pivotY="50%"
    android:duration="3000"
    android:repeatCount="infinite"
    android:repeatMode="reverse" />
```

Logic behind the start-up

Regarding the logic behind the start-up activity, we'll simply create an animation object inside the onCreate method which will load the animation we've created inside the anim folder and give it to the image button we've selected. To start the animation, we call the method startAnimation, giving as parameter the animation object, we just created.

To continue with the next view, we have to add a method which starts a new activity for us. We'll call this method launchApp. Inside this method we need to create a new intent with the current context and the new activity class we want to launch. After completing those steps, we need to start the activity with the intent we just created.



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

    setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);

    ImageButton imageButton = findViewById(R.id.imageButton);
    Animation animation = AnimationUtils.loadAnimation( context: this, R.anim.bounce);
    imageButton.startAnimation(animation);
}
```

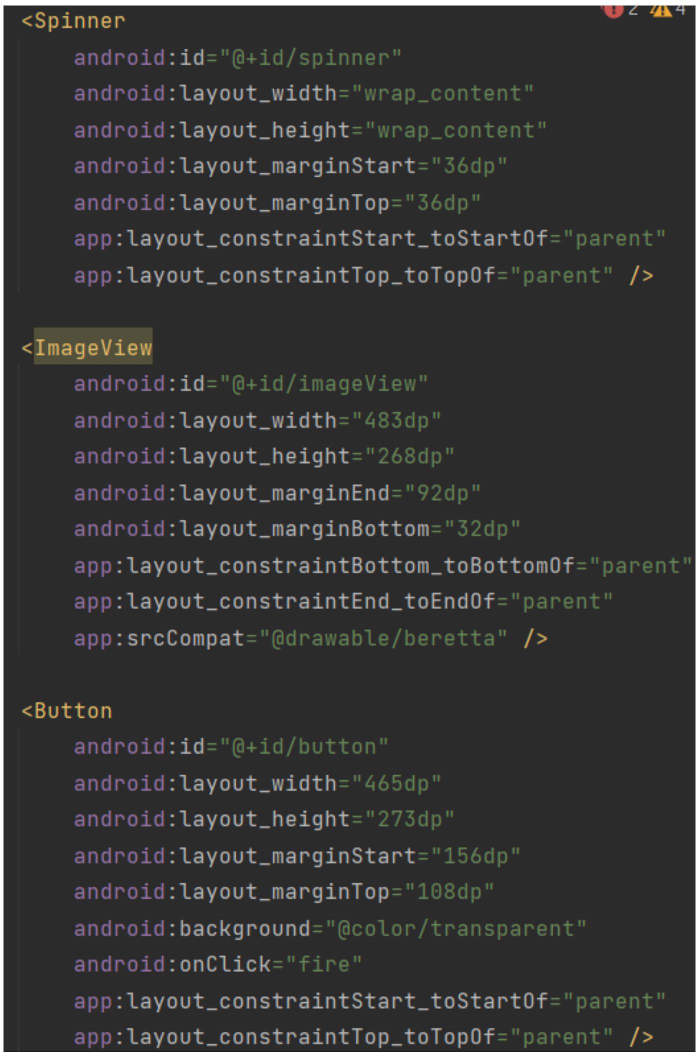
Shoot activity

This activity will be the hardest one we've designed so far. This layout includes an image view where we'll display the image of the gun itself, a button overlaying the image which will trigger the shooting action, and a spinner which will act as a menu for swapping the weapons.

At first, we set the background to the one provided in the files, a nice carbon-fiber texture. To start placing the elements we'll first create the image view and set it's constraining. Set this element to a default of a gun you want, for this project I decided to put the beretta as the default one which will be displayed when the app starts.

We then create an overlapping button and place it above the image view. Lastly, we place a spinner element preferably on the top left corner.

With this the layout is completed and we can pass on to the logic behind the app.



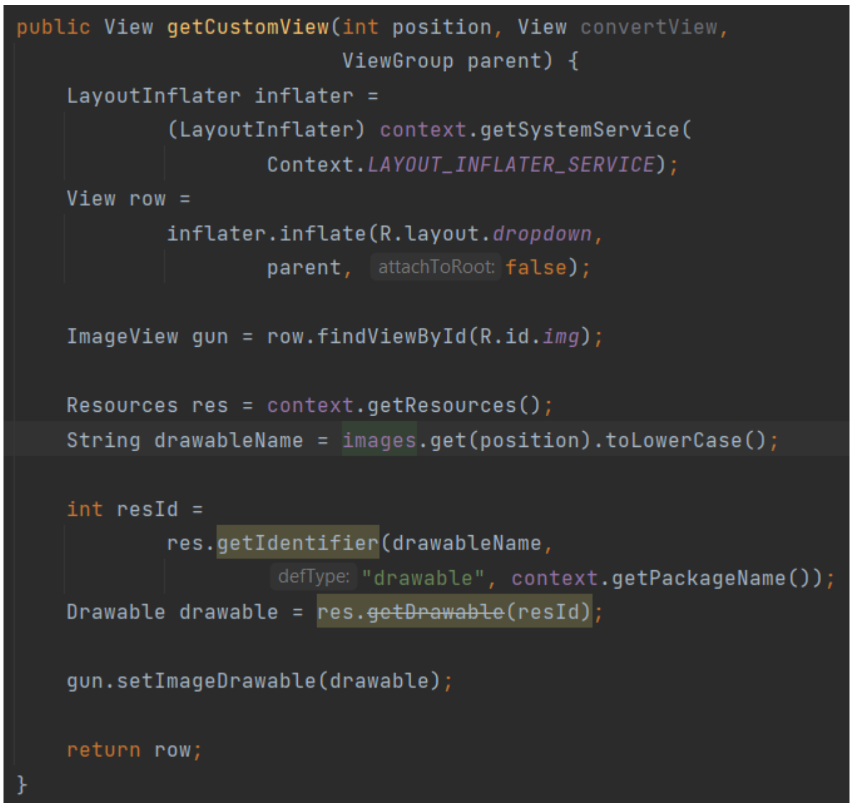
```xml
<Spinner
    android:id="@+id/spinner"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="36dp"
    android:layout_marginTop="36dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<ImageView
    android:id="@+id/imageView"
    android:layout_width="483dp"
    android:layout_height="268dp"
    android:layout_marginEnd="92dp"
    android:layout_marginBottom="32dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:srcCompat="@drawable/beretta" />

<Button
    android:id="@+id/button"
    android:layout_width="465dp"
    android:layout_height="273dp"
    android:layout_marginStart="156dp"
    android:layout_marginTop="108dp"
    android:background="@color/transparent"
    android:onClick="fire"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Logic behind the shoot activity

To kick off this activity we need to assign every element created to a variable so we can access them and then create a list of strings which will hold the names of the images representing our guns.

Setting up the spinner adapter comes next; this process is probably the most complex part of this application. Here we need to create a separate spinner adapter class which will act as a blueprint for our custom-made spinner adapter.

To explain what exactly this is; this spinner adapter is the one we use to create a dropdown menu, display it, and select elements from it. In essence we'll be using two of the already implemented methods: getDrowDownView and getView.

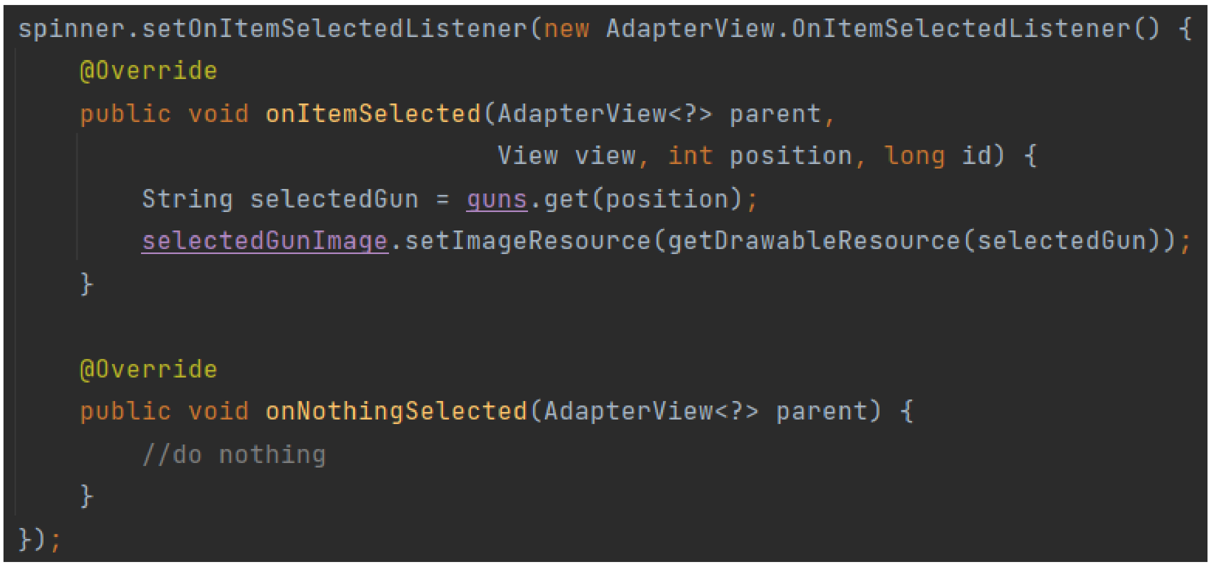


```java
public View getCustomView(int position, View convertView,
                          ViewGroup parent) {
    LayoutInflater inflater =
            (LayoutInflater) context.getSystemService(
                    Context.LAYOUT_INFLATER_SERVICE);
    View row =
            inflater.inflate(R.layout.dropdown,
                    parent,  attachToRoot: false);

    ImageView gun = row.findViewById(R.id.img);

    Resources res = context.getResources();
    String drawableName = images.get(position).toLowerCase();

    int resId =
            res.getIdentifier(drawableName,
                     defType: "drawable", context.getPackageName());
    Drawable drawable = res.~~getDrawable~~(resId);

    gun.setImageDrawable(drawable);

    return row;
}
```

The methods are presented in the exemplified code. The getDropDownView method handles displaying the entire dropdown menu row by row by the usage of the custom-made method showed in the image.

Essentially this method handles creation and positioning of an entire row. By repeatedly calling this method we create the dropdown menu row by row.

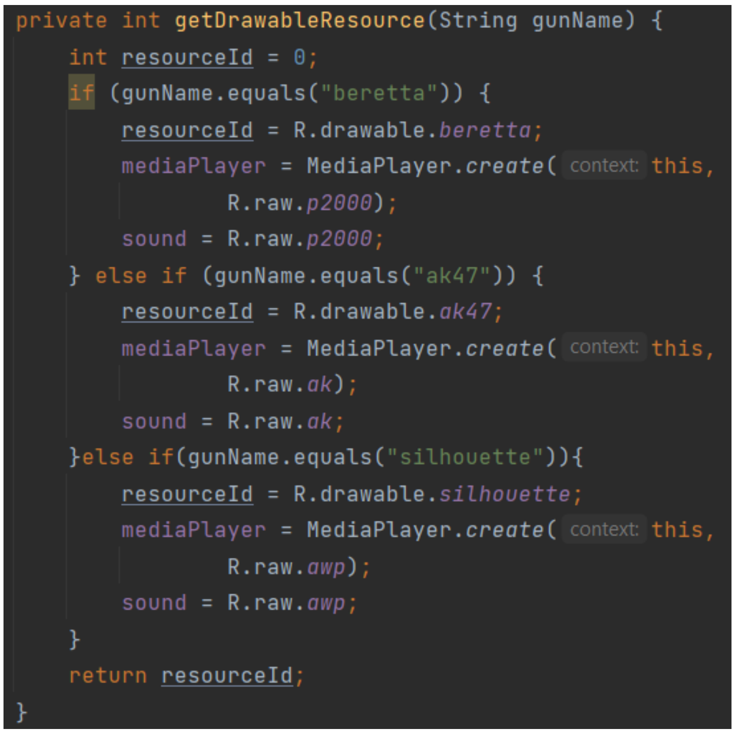
Using the getView method we remember the last row which was clicked and display it as a placeholder. For example, when using the ak, after clicking on it the menu will show we're currently using the ak.



```java
spinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
    @Override
    public void onItemSelected(AdapterView<?> parent,
                               View view, int position, long id) {
        String selectedGun = guns.get(position);
        selectedGunImage.setImageResource(getDrawableResource(selectedGun));
    }

    @Override
    public void onNothingSelected(AdapterView<?> parent) {
        //do nothing
    }
});
```

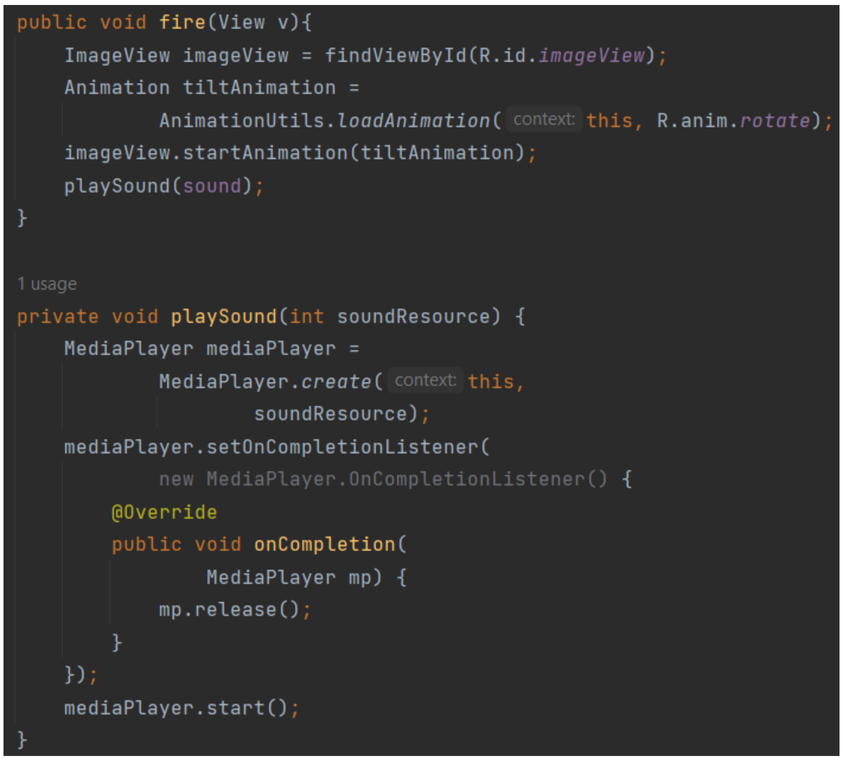
As showed in the image above, by using the spinner adapter we get the current gun from the list by position provided by the adapter, then we set the image view via the custom-made method getDrawableResource.

To explain this method, it gets the gun name as parameter and checks which one of them it is, when we find which gun we need to display we set the id to the id of the image and assign the media player their respective sounds which will be used in a later method to play the sound of the shot.



```java
private int getDrawableResource(String gunName) {
    int resourceId = 0;
    if (gunName.equals("beretta")) {
        resourceId = R.drawable.beretta;
        mediaPlayer = MediaPlayer.create( context: this,
                R.raw.p2000);
        sound = R.raw.p2000;
    } else if (gunName.equals("ak47")) {
        resourceId = R.drawable.ak47;
        mediaPlayer = MediaPlayer.create( context: this,
                R.raw.ak);
        sound = R.raw.ak;
    }else if(gunName.equals("silhouette")){
        resourceId = R.drawable.silhouette;
        mediaPlayer = MediaPlayer.create( context: this,
                R.raw.awp);
        sound = R.raw.awp;
    }
    return resourceId;
}
```

For the fire and playSound methods we'll take a simple approach. The fire method will load and start a tilting or rotation animation which will tilt the tip of the gun upwards to create the illusion of a real shot. We call the playSound method to play the sound set above via the getDrawableResource. We create a media player object with the respective sound and on we start it after creation. We create a branch which will handle the deletion of the media player after the sound was completed.



```java
public void fire(View v){
    ImageView imageView = findViewById(R.id.imageView);
    Animation tiltAnimation =
            AnimationUtils.loadAnimation( context: this, R.anim.rotate);
    imageView.startAnimation(tiltAnimation);
    playSound(sound);
}

1 usage
private void playSound(int soundResource) {
    MediaPlayer mediaPlayer =
            MediaPlayer.create( context: this,
                    soundResource);
    mediaPlayer.setOnCompletionListener(
            new MediaPlayer.OnCompletionListener() {
        @Override
        public void onCompletion(
                MediaPlayer mp) {
            mp.release();
        }
    });
    mediaPlayer.start();
}
```

With this the application should be completed after assigning the methods.

Hard application: Reminder

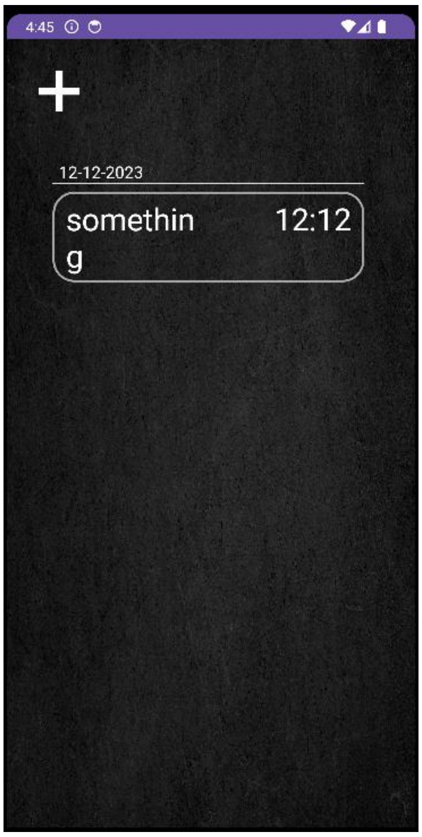
5.1 Overview

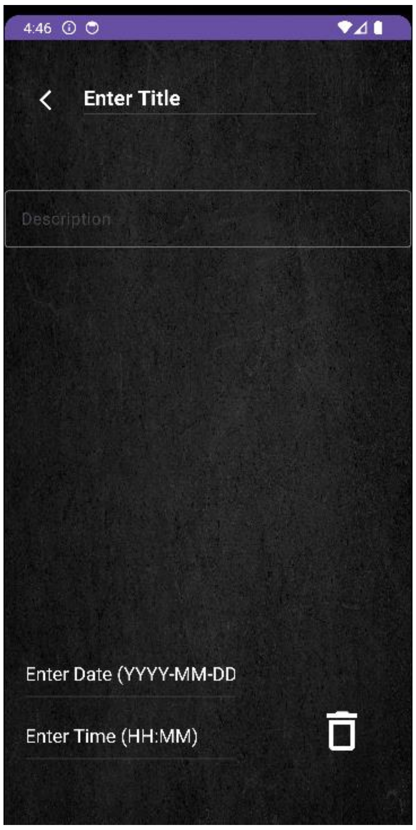
Finally, we reached the final phase of this tutorial but unfortunately it is the hardest one so far. This last part will focus on the development of the hard application which will be a reminder. With this reminder app we can set notes with different titles, descriptions, dates, and times, which will trigger notifications. It will consist of multiple views and shared preferences.

5.2 Key Features

The key feature of this application is the fact it can be used most of the times for scheduling and planning. It gives the opportunity to edit and delete unwanted or wrongly written notes, has checks to ensure the dates are correct. In this last tutorial we'll be learning how shared preferences work and how to set up notifications.

The date and time will be set in local type to Europe format. The title, date and time are mandatory fields while the description can be omitted if need.





5.3 Development

To kick off the development process properly we first need to address the permissions inside the manifest file. We need to set the following permissions to be able to display the notifications and manage them.

We also need to create a receiver with the name of AlarmReceiver to handle the triggering of the notificaions.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.POST_NOTIFICATIONS" />
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.USE_EXACT_ALARM" />
```

Main activity

For the main activity layout, we need to create an image button which will act as the add button for new notes, a scroll view and a linear layout.

The scroll view contains all the notes which will be represented as different rows showing the title and time it's supposed to fire a notification at.

As expected every element needs specific constraints.

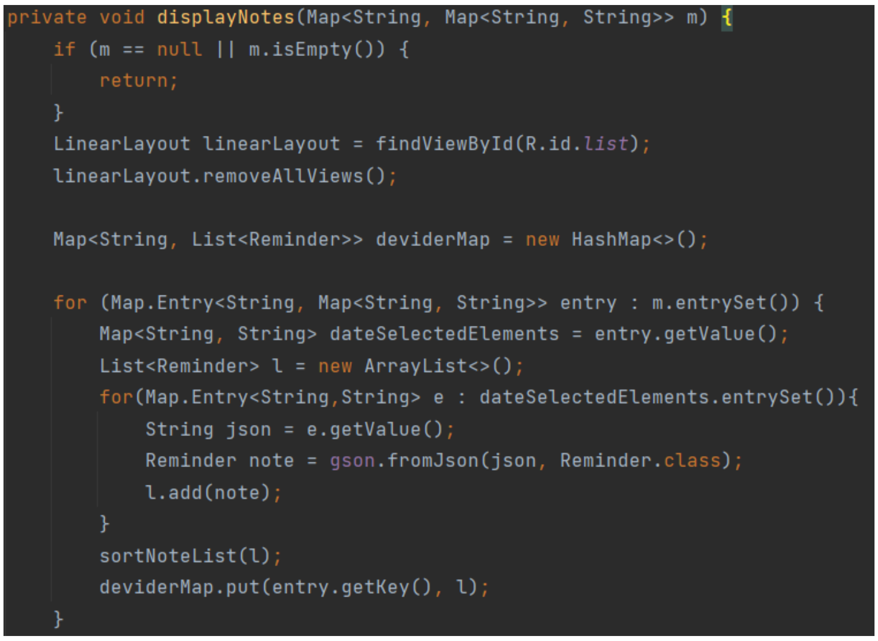
```
<ImageButton
    android:id="@+id/floatingActionButton2"
    android:layout_width="40dp"
    android:layout_height="40dp"
    android:layout_marginStart="30dp"
    android:layout_marginTop="30dp"
    android:onClick="launchActivity"
    android:background="?attr/selectableItemBackgroundBorderless"
    android:src="@drawable/plus"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<ScrollView
    android:layout_width="300dp"
    android:layout_height="641dp"
    android:layout_marginBottom="4dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.473"
    app:layout_constraintStart_toStartOf="parent">

    <LinearLayout
        android:id="@+id/list"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical" />
</ScrollView>
```

Logic behind Main Activity

At the start of the main activity, we need to set up a GSON parser which will be used for the data since the communication between activities is made through shared preferences. The shared preferences need to be told which of the data to transfer to each other. For this we need to encrypt them into json and parse them when reading in the necessary place.

To store all the data, we use a map with keys being strings and values different maps consisting of strings only.

To start off we have implemented the displayNotes method which is the main method used to display the notes we have stored inside the map.

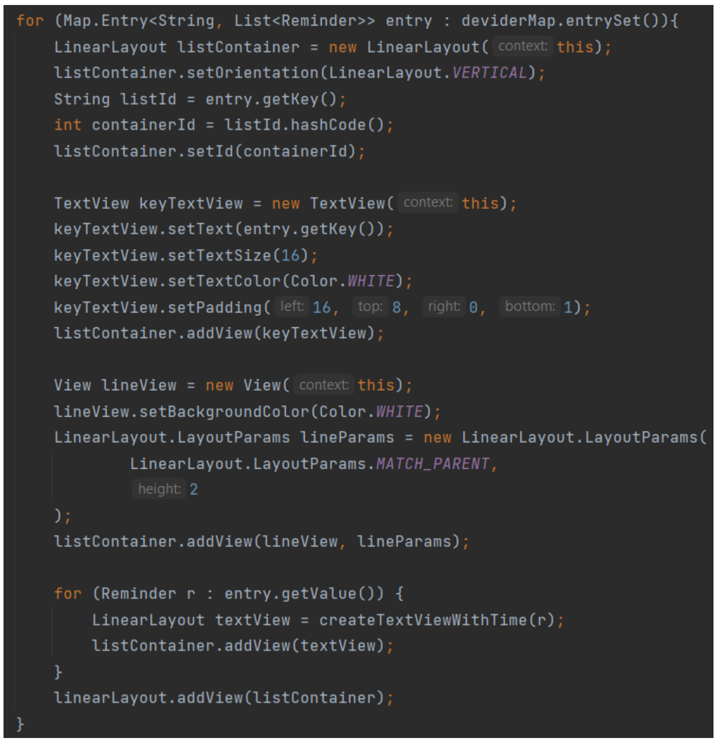


```
private void displayNotes(Map<String, Map<String, String>> m) {
    if (m == null || m.isEmpty()) {
        return;
    }
    LinearLayout linearLayout = findViewById(R.id.list);
    linearLayout.removeAllViews();

    Map<String, List<Reminder>> deviderMap = new HashMap<>();

    for (Map.Entry<String, Map<String, String>> entry : m.entrySet()) {
        Map<String, String> dateSelectedElements = entry.getValue();
        List<Reminder> l = new ArrayList<>();
        for(Map.Entry<String,String> e : dateSelectedElements.entrySet()){
            String json = e.getValue();
            Reminder note = gson.fromJson(json, Reminder.class);
            l.add(note);
        }
        sortNoteList(l);
        deviderMap.put(entry.getKey(), l);
    }
```

In the beginning we check to see if the map is empty or not, in case it is empty we won't be displaying anything. After we pass through all the values from the map and sort them together via the dates they belong to in clusters for aesthetics and simplicity. We convert the json formatted objects into reminder types so we can access them easier and more directly.

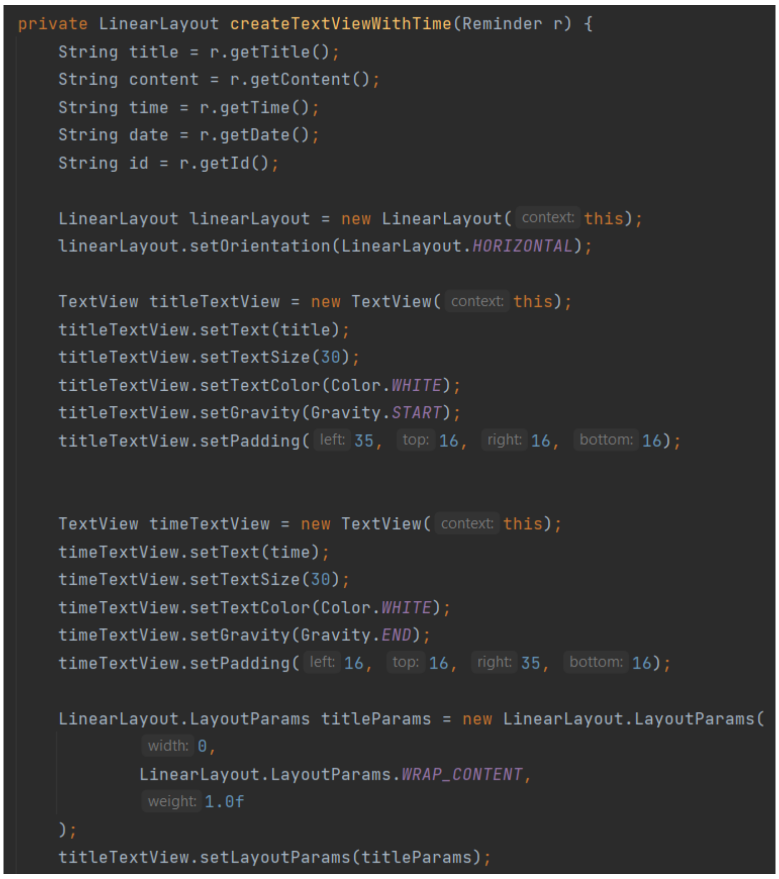
For the next phase of the display method, we need to loop through all the values from the newly created map. In here we'll be creating a special divider for each date, after the line is created, we need to create a different linear layout for each date in which we'll be storing the specific notes with those specific dates. This was thought to allow dynamic insertion and deletion.

After the individual layouts are created, we loop through the list of notes corresponding to that date and create a view by using the createTextViewWithTime method, which takes a reminder as parameter and converts it into a row displayed in the overview section. After the rows were created the whole linear layout is inserted in the scrollable layout.



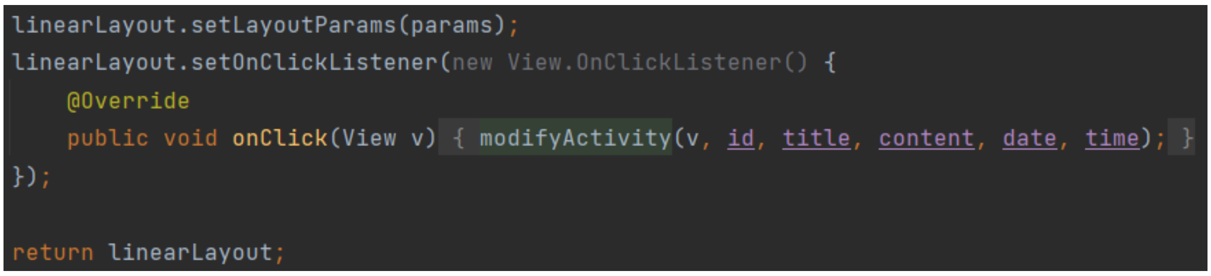
```java
for (Map.Entry<String, List<Reminder>> entry : deviderMap.entrySet()){
    LinearLayout listContainer = new LinearLayout( context: this);
    listContainer.setOrientation(LinearLayout.VERTICAL);
    String listId = entry.getKey();
    int containerId = listId.hashCode();
    listContainer.setId(containerId);

    TextView keyTextView = new TextView( context: this);
    keyTextView.setText(entry.getKey());
    keyTextView.setTextSize(16);
    keyTextView.setTextColor(Color.WHITE);
    keyTextView.setPadding( left: 16, top: 8, right: 0, bottom: 1);
    listContainer.addView(keyTextView);

    View lineView = new View( context: this);
    lineView.setBackgroundColor(Color.WHITE);
    LinearLayout.LayoutParams lineParams = new LinearLayout.LayoutParams(
            LinearLayout.LayoutParams.MATCH_PARENT,
             height: 2
    );
    listContainer.addView(lineView, lineParams);

    for (Reminder r : entry.getValue()) {
        LinearLayout textView = createTextViewWithTime(r);
        listContainer.addView(textView);
    }
    linearLayout.addView(listContainer);
}
```

As one can see in the following image the implementation of the views is rather easy, we need to get all the data required then create a linear layout in which we'll introduce our data step by step, at first the title then the time, the first one floating at the left side and the second at the right off the container to leave a good impression on the user and improve readability.



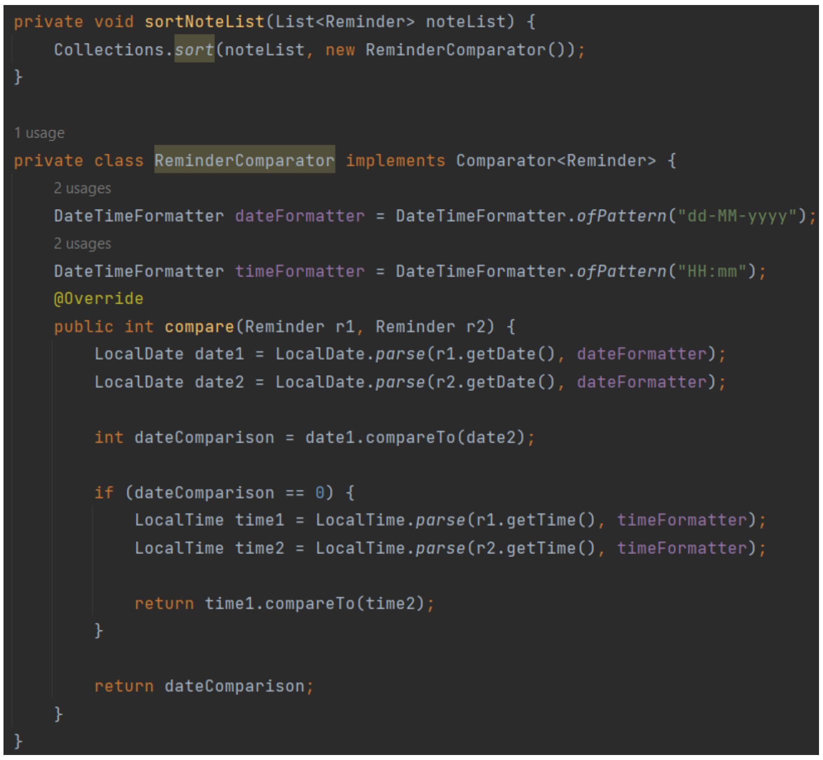
```java
private LinearLayout createTextViewWithTime(Reminder r) {
    String title = r.getTitle();
    String content = r.getContent();
    String time = r.getTime();
    String date = r.getDate();
    String id = r.getId();

    LinearLayout linearLayout = new LinearLayout( context: this);
    linearLayout.setOrientation(LinearLayout.HORIZONTAL);

    TextView titleTextView = new TextView( context: this);
    titleTextView.setText(title);
    titleTextView.setTextSize(30);
    titleTextView.setTextColor(Color.WHITE);
    titleTextView.setGravity(Gravity.START);
    titleTextView.setPadding( left: 35, top: 16, right: 16, bottom: 16);


    TextView timeTextView = new TextView( context: this);
    timeTextView.setText(time);
    timeTextView.setTextSize(30);
    timeTextView.setTextColor(Color.WHITE);
    timeTextView.setGravity(Gravity.END);
    timeTextView.setPadding( left: 16, top: 16, right: 35, bottom: 16);

    LinearLayout.LayoutParams titleParams = new LinearLayout.LayoutParams(
             width: 0,
            LinearLayout.LayoutParams.WRAP_CONTENT,
             weight: 1.0f
    );
    titleTextView.setLayoutParams(titleParams);
```

At the end of the method, we need to add a click event which will call the modification method in case we want to reopen the note and modify it. This modifyActivity method essentially functions exactly like a launching activity where we create a new activity but in here, we set and send the data provided as parameters in order to set the note to the current values, such as title and so on.



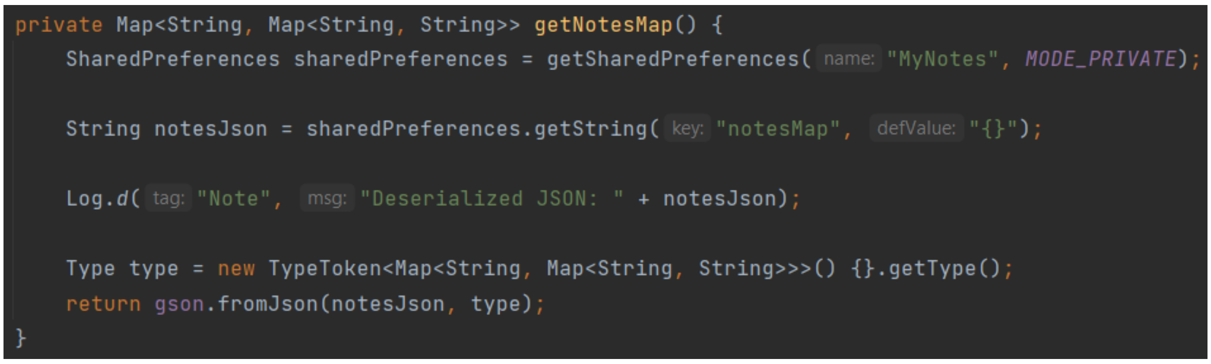
```java
linearLayout.setLayoutParams(params);
linearLayout.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) { modifyActivity(v, id, title, content, date, time); }
});

return linearLayout;
```

In order to sort the notes we need to implement the following method which will make use of the collections basic sorting algorithm, were we'll give our list and a way to arrange our items. The way to arrange the items is implemented as a comparator which takes the two formats and creates a comparing method which takes our initial dates comparing them between each other and giving back a reply depending on the result, either rearrange or keep the items in the same order.



```java
private void sortNoteList(List<Reminder> noteList) {
    Collections.sort(noteList, new ReminderComparator());
}

1 usage
private class ReminderComparator implements Comparator<Reminder> {
    2 usages
    DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("dd-MM-yyyy");
    2 usages
    DateTimeFormatter timeFormatter = DateTimeFormatter.ofPattern("HH:mm");
    @Override
    public int compare(Reminder r1, Reminder r2) {
        LocalDate date1 = LocalDate.parse(r1.getDate(), dateFormatter);
        LocalDate date2 = LocalDate.parse(r2.getDate(), dateFormatter);

        int dateComparison = date1.compareTo(date2);

        if (dateComparison == 0) {
            LocalTime time1 = LocalTime.parse(r1.getTime(), timeFormatter);
            LocalTime time2 = LocalTime.parse(r2.getTime(), timeFormatter);

            return time1.compareTo(time2);
        }

        return dateComparison;
    }
}
```

To work with preferences, we'll need to make use of this method which will be sort of redundant because almost every functionality in this app needs to make use of such a method. In this method we simply fetch the data from our preferences and parse it using json to a map type we already predefined. Take note this is one of the core functionalities of this app and needs to be implemented correctly, otherwise the data will not be accessed properly.



```java
private Map<String, Map<String, String>> getNotesMap() {
    SharedPreferences sharedPreferences = getSharedPreferences( name: "MyNotes", MODE_PRIVATE);

    String notesJson = sharedPreferences.getString( key: "notesMap",  defValue: "{}");

    Log.d( tag: "Note",  msg: "Deserialized JSON: " + notesJson);

    Type type = new TypeToken<Map<String, Map<String, String>>>() {}.getType();
    return gson.fromJson(notesJson, type);
}
```

To ensure the dynamic behaviour of the app, we have implemented a resume method which takes advantage of the already implemented display method to rearrange the items.

Note layout

The notes layout is simple, we have different items and text fields for example: a back button which will act as a save button, this will implement the save functionality and the return to the main view, a title field, a date field, a time field, a description input field, and a delete button.

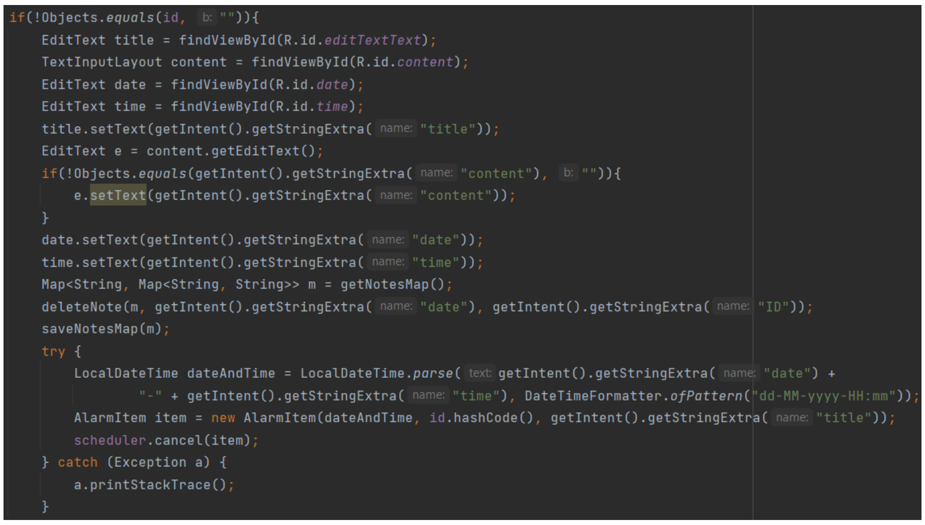
Nothing special is happening in the layouts now, we just need to set the hints for the description properly and their constrains as well as the proper ids which should be checked to ensure their validity.

The buttons are image buttons and the necessary resources for the icons can be found inside the drawable folder. We also make use of icons we created as xml files like the border element we already used before so you can look at how I made this arrow or just use a built-in icon.

Logic behind the Note Activity

To start of the note activity, we need to create the GSON parser and a scheduler which will be implemented at a later time.

For the creation method we have two branches that can be accessed, at first if we don’t have to modify an already existing one we can simply continue with our creation process, if the case involves already received data and a modification process we need to address that case properly.



```java
if(!Objects.equals(id, b: "")){
    EditText title = findViewById(R.id.editTextText);
    TextInputLayout content = findViewById(R.id.content);
    EditText date = findViewById(R.id.date);
    EditText time = findViewById(R.id.time);
    title.setText(getIntent().getStringExtra( name: "title"));
    EditText e = content.getEditText();
    if(!Objects.equals(getIntent().getStringExtra( name: "content"), b: "")){
        e.setText(getIntent().getStringExtra( name: "content"));
    }
    date.setText(getIntent().getStringExtra( name: "date"));
    time.setText(getIntent().getStringExtra( name: "time"));
    Map<String, Map<String, String>> m = getNotesMap();
    deleteNote(m, getIntent().getStringExtra( name: "date"), getIntent().getStringExtra( name: "ID"));
    saveNotesMap(m);
    try {
        LocalDateTime dateAndTime = LocalDateTime.parse( text: getIntent().getStringExtra( name: "date") +
                "-" + getIntent().getStringExtra( name: "time"), DateTimeFormatter.ofPattern("dd-MM-yyyy-HH:mm"));
        AlarmItem item = new AlarmItem(dateAndTime, id.hashCode(), getIntent().getStringExtra( name: "title"));
        scheduler.cancel(item);
    } catch (Exception a) {
        a.printStackTrace();
    }
```

in the showed image data is checked and set if it exists to the corresponding fields. While the data is set the scheduler goes ahead and shuts down the already scheduled alarm set on the time and date received. The time will be rewritten, and a new alarm will be set in case the modified note is saved and not deleted.

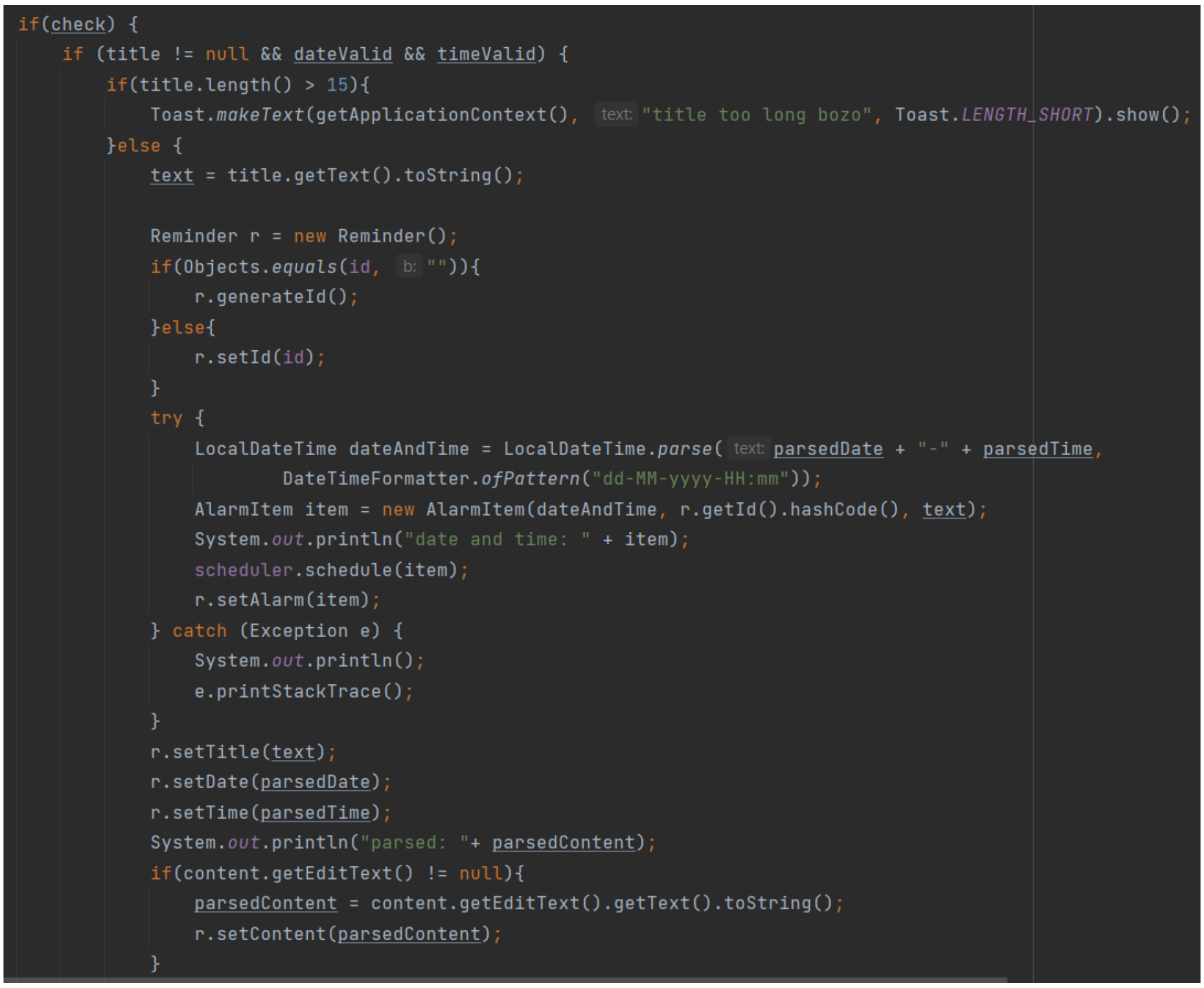
We'll be now passing to the back method which acts as the method triggered by the arrow button and saves the newly created note. In the beginning we'll be setting variables and perform checks on the input data especially for the time, date and title which are mandatory.

If the checks pass then we can create a reminder type object and set it's data to the input taken from the layout, we need to create a new special id which will act as a way to recognize the reminder we just created. The reminder is a different class which just holds the necessary information, reason for what we don't need to show it especially since it is a simple object containing the right fields.

After the reminder is set we call the saveNote method which is basically just a way to store the note inside a map which was previously showed, this method makes use of the map and then parses it by usage of the previously mentioned GSON parser.

After the map is updated and parsed into json stored in the shared preferences we can proceed to finish the activity and return to the main one.

This class contains different methods to work with the map, deletion and save methods but also methods to check time and validate specific data.



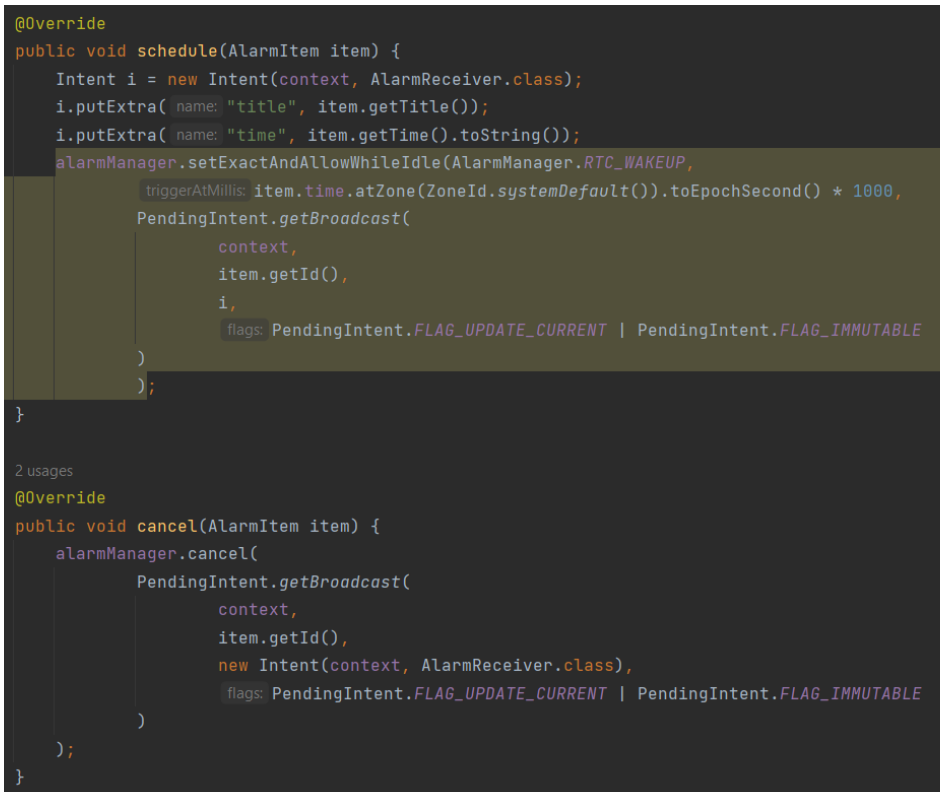
```
if(check) {
    if (title != null && dateValid && timeValid) {
        if(title.length() > 15){
            Toast.makeText(getApplicationContext(),  text: "title too long bozo", Toast.LENGTH_SHORT).show();
        }else {
            text = title.getText().toString();

            Reminder r = new Reminder();
            if(Objects.equals(id,  b: "")){
                r.generateId();
            }else{
                r.setId(id);
            }
            try {
                LocalDateTime dateAndTime = LocalDateTime.parse( text: parsedDate + "-" + parsedTime,
                        DateTimeFormatter.ofPattern("dd-MM-yyyy-HH:mm"));
                AlarmItem item = new AlarmItem(dateAndTime, r.getId().hashCode(), text);
                System.out.println("date and time: " + item);
                scheduler.schedule(item);
                r.setAlarm(item);
            } catch (Exception e) {
                System.out.println();
                e.printStackTrace();
            }
            r.setTitle(text);
            r.setDate(parsedDate);
            r.setTime(parsedTime);
            System.out.println("parsed: "+ parsedContent);
            if(content.getEditText() != null){
                parsedContent = content.getEditText().getText().toString();
                r.setContent(parsedContent);
            }
```

Logic behind the Scheduler

The notifications are handled in a more complex way, firstly we set up the Scheduler inside the note class. This scheduler is used to handle and manage the alarms.

The AndroidAlarmScheduler class implements the AlarmScheduler interface which has two specific methods: schedule and delete. As the name suggests, the schedule method takes in an AlarmItem parameter which is like a reminder, but this one only takes the information which needs to be displayed when the notification pops up, in our case being the title and time of the reminder.

The schedule method sets up the time for triggering the notification by the usage of a pending intent. In our case we make it able to fire off alarms even if the phone isn't active with the app open, feature which is needed for a reminder like app.



```java
@Override
public void schedule(AlarmItem item) {
    Intent i = new Intent(context, AlarmReceiver.class);
    i.putExtra( name: "title", item.getTitle());
    i.putExtra( name: "time", item.getTime().toString());
    alarmManager.setExactAndAllowWhileIdle(AlarmManager.RTC_WAKEUP,
             triggerAtMillis: item.time.atZone(ZoneId.systemDefault()).toEpochSecond() * 1000,
            PendingIntent.getBroadcast(
                    context,
                    item.getId(),
                    i,
                     flags: PendingIntent.FLAG_UPDATE_CURRENT | PendingIntent.FLAG_IMMUTABLE
            )
            );
}

2 usages
@Override
public void cancel(AlarmItem item) {
    alarmManager.cancel(
            PendingIntent.getBroadcast(
                    context,
                    item.getId(),
                    new Intent(context, AlarmReceiver.class),
                     flags: PendingIntent.FLAG_UPDATE_CURRENT | PendingIntent.FLAG_IMMUTABLE
            )
    );
}
```

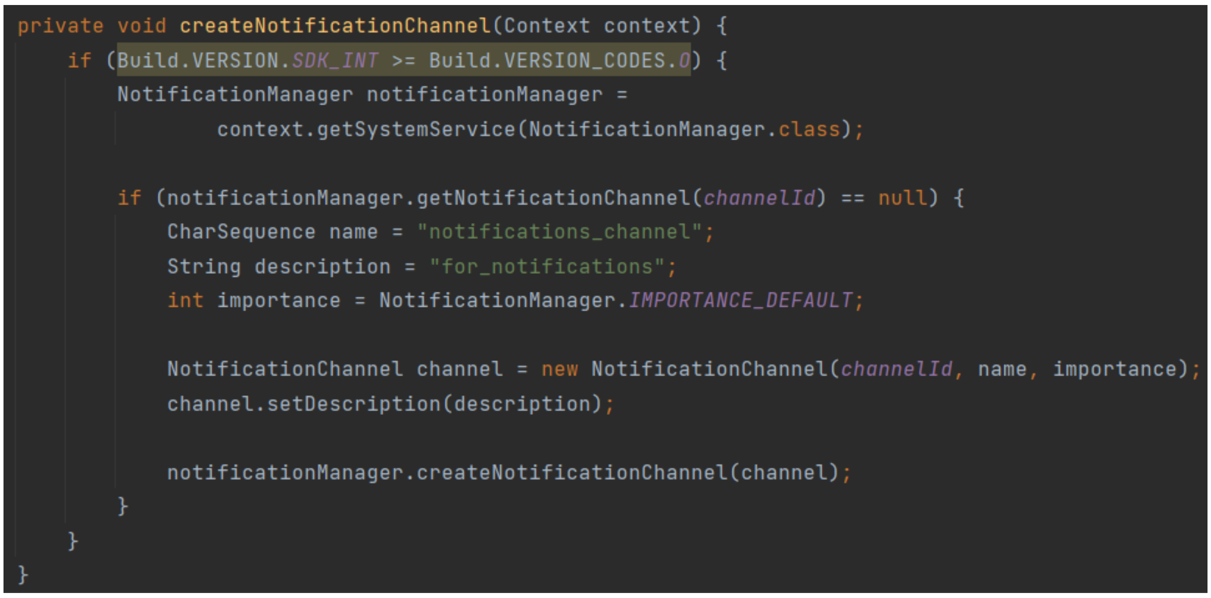
The cancel method simply takes the id of the alarm we have already set and unbinds it or in fewer words deletes it rendering it unable to fire.

These two methods are the core of our alarms since by using this set alarm we need to create the notification themselves. The notifications are a separate process, they are just messages which need to appear on screen at the needed time. This class we just implemented takes care of the scheduling of such notifications. Now we need to set the notification channel and the way notifications should look, then call them when the time is necessary.

The receiver was already declared in the beginning of the development process, but it is required for the scheduler to take effect.

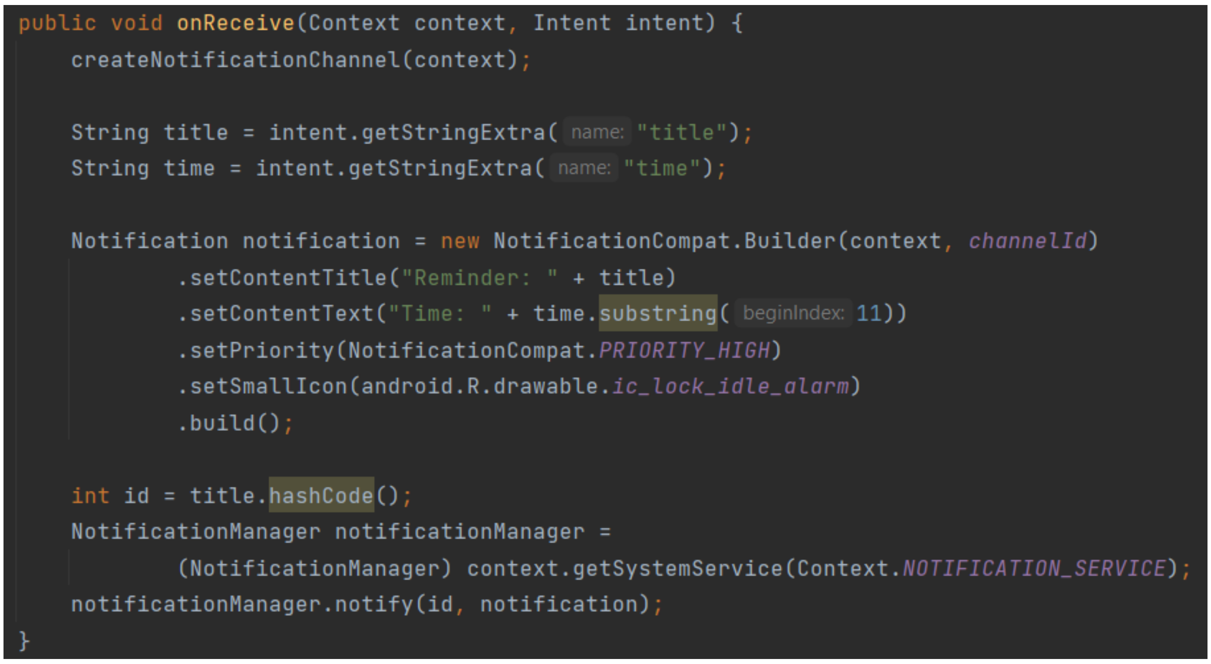
Logic behind the notifications

To be able to work with notifications at first, we need to set up a notification channel. Usually this notification channels need to have an unique id in case we need to display different types of messages, but since we'll be displaying only a single type we can declare a static id which will be used to create the notification channel, once created this notification channel won't be recreated but only used so we can set it up once and forget about it for the rest of the development process.



```java
private void createNotificationChannel(Context context) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        NotificationManager notificationManager =
                context.getSystemService(NotificationManager.class);

        if (notificationManager.getNotificationChannel(channelId) == null) {
            CharSequence name = "notifications_channel";
            String description = "for_notifications";
            int importance = NotificationManager.IMPORTANCE_DEFAULT;

            NotificationChannel channel = new NotificationChannel(channelId, name, importance);
            channel.setDescription(description);

            notificationManager.createNotificationChannel(channel);
        }
    }
}
```

For the last phase of our development process, we need to set up the notification themselves. Here we make use of the onReceive method which essentially works as a bullet, the alarm trigger fires on time and shots the notification when the signal is received. We simply create a notification, give it an id, a title, a time field, set its priority to high and an icon to the basic lock icon. We then make use of the notify method to trigger the created notification.



```java
public void onReceive(Context context, Intent intent) {
    createNotificationChannel(context);

    String title = intent.getStringExtra( name: "title");
    String time = intent.getStringExtra( name: "time");

    Notification notification = new NotificationCompat.Builder(context, channelId)
            .setContentTitle("Reminder: " + title)
            .setContentText("Time: " + time.substring( beginIndex: 11))
            .setPriority(NotificationCompat.PRIORITY_HIGH)
            .setSmallIcon(android.R.drawable.ic_lock_idle_alarm)
            .build();

    int id = title.hashCode();
    NotificationManager notificationManager =
            (NotificationManager) context.getSystemService(Context.NOTIFICATION_SERVICE);
    notificationManager.notify(id, notification);
}
```

With this the major functionality of the app is done, more details can be found in the code section provided within the project, but this tutorial should have touched all the important points of the application.

Conclusions

6.1 Lessons Learned

This project showcases the versatility and practical application of Java in developing functional and user-friendly applications. The tutorial-style documentation aims to empower users and developers to grasp the concepts behind each application and adapt them for their specific needs.

Each of the applications selected for this tutorial were stepping stones and each introduced new concepts.

The first app introduced us to the IDE and some basic concepts about views, contexts, layouts and constrains.

The second application took the layout elements a step further focusing mostly on layout elements, animations, drawable elements and icons all while diving deeper into logistics and fundamental features like spinner adapters and menus.

Finally, the third application utilized all the things learned so far and combined them into a nice usable project which focuses both on dynamic layouts, multiple views and shared preferences while also introducing a core concept of mobile development, notifications, and schedulers.

In conclusion, this tutorial was a fun experience. I also learned a lot from this project, I hope the information provided was properly explained and utilized in a good enough manor able to keep you entertained and hungry for knowledge.