ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

READER NHS3152 APP DOCUMENTATION

Originator:

Function	Nome
Project Manager	Leo Torchia
Project Programmer (internship)	Alessandro Rossi

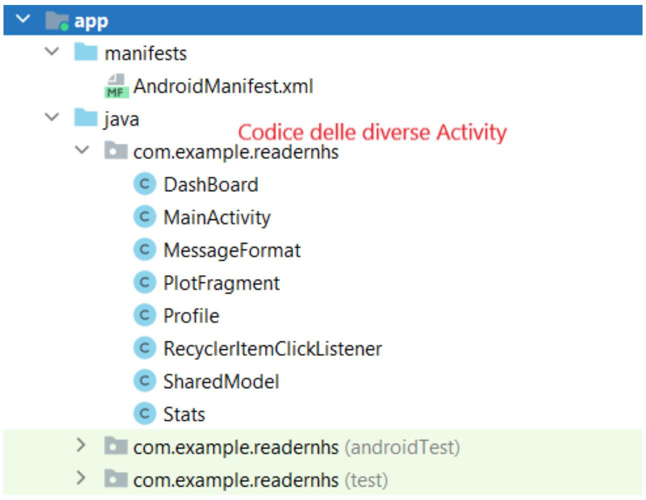
Version:

Person	Date
Leo Torchia	2022/01/03

Contents

Project Structure

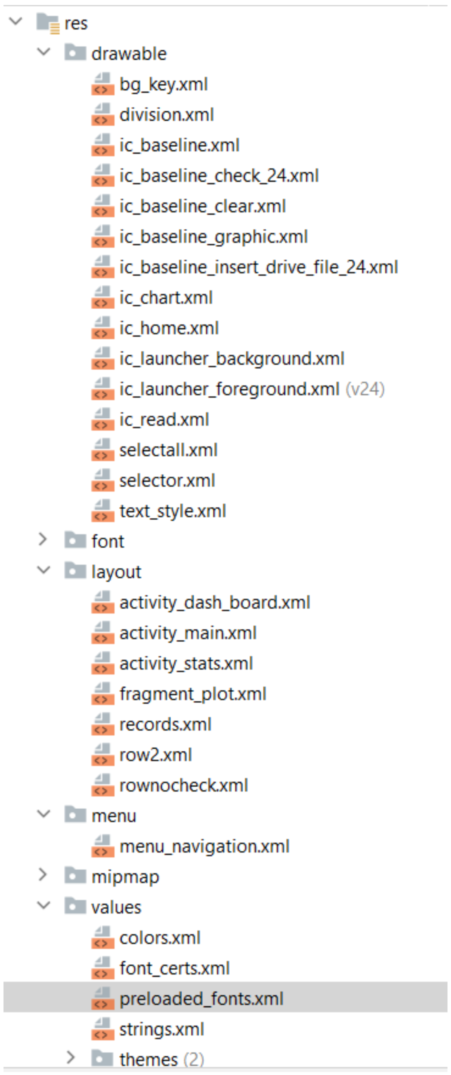
The project logic (main) is in *java/com.example.readernhs/*. All graphic resources are in the folder res.



```
app
  manifests
    AndroidManifest.xml
  java
    com.example.readernhs
      DashBoard
      MainActivity
      MessageFormat
      PlotFragment
      Profile
      RecyclerItemClickListener
      SharedModel
      Stats
    com.example.readernhs (androidTest)
    com.example.readernhs (test)
```

There most important folders are:

Folder res/..	Function
Drawable	All Icons and graphic elements
Layout	All activity layouts
Menu	Lower menu of Application
Values	Contains the 3 main files:
Values/**Colors.xml**	Application colors specified
Values/**certs.xmls** Value/**preloadedfonts.xml**	Application fonts specified
Values/**string.xml**	Application strings specified

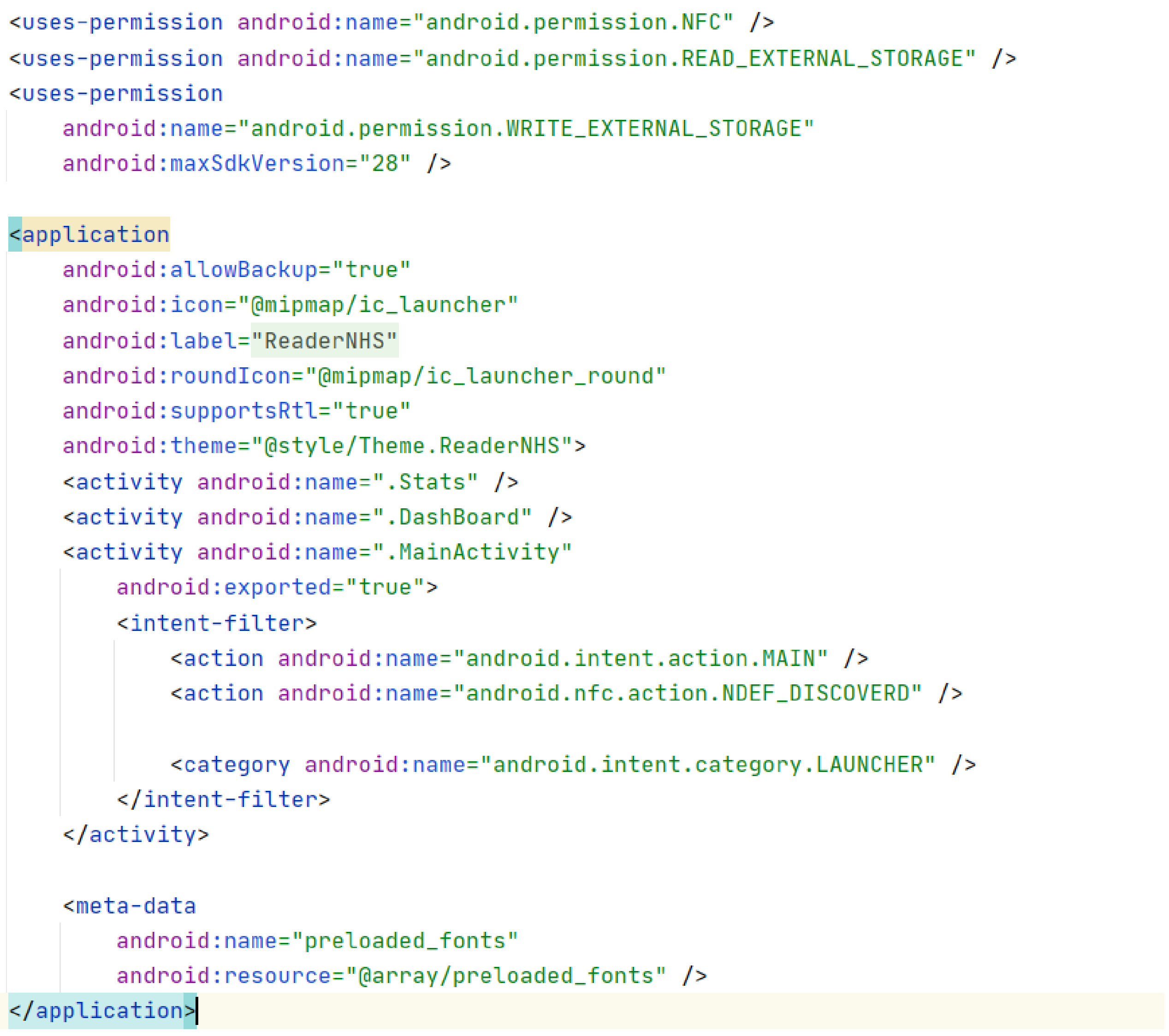


```
res
  drawable
    bg_key.xml
    division.xml
    ic_baseline.xml
    ic_baseline_check_24.xml
    ic_baseline_clear.xml
    ic_baseline_graphic.xml
    ic_baseline_insert_drive_file_24.xml
    ic_chart.xml
    ic_home.xml
    ic_launcher_background.xml
    ic_launcher_foreground.xml (v24)
    ic_read.xml
    selectall.xml
    selector.xml
    text_style.xml
  font
  layout
    activity_dash_board.xml
    activity_main.xml
    activity_stats.xml
    fragment_plot.xml
    records.xml
    row2.xml
    rownocheck.xml
  menu
    menu_navigation.xml
  mipmap
  values
    colors.xml
    font_certs.xml
    preloaded_fonts.xml
    strings.xml
    themes (2)
```

Application Launch

When launched in the file *AndroidManifest.xml* we check:

- The application
- The activities
- The permissions
- The intent filters

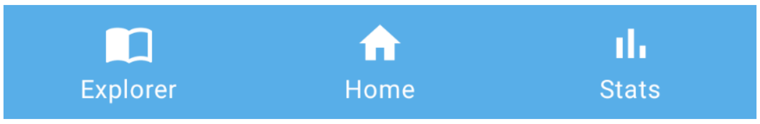


```xml
<uses-permission android:name="android.permission.NFC" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE"
    android:maxSdkVersion="28" />

<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="ReaderNHS"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.ReaderNHS">
    <activity android:name=".Stats" />
    <activity android:name=".DashBoard" />
    <activity android:name=".MainActivity"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <action android:name="android.nfc.action.NDEF_DISCOVERD" />

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

    <meta-data
        android:name="preloaded_fonts"
        android:resource="@array/preloaded_fonts" />
</application>
```

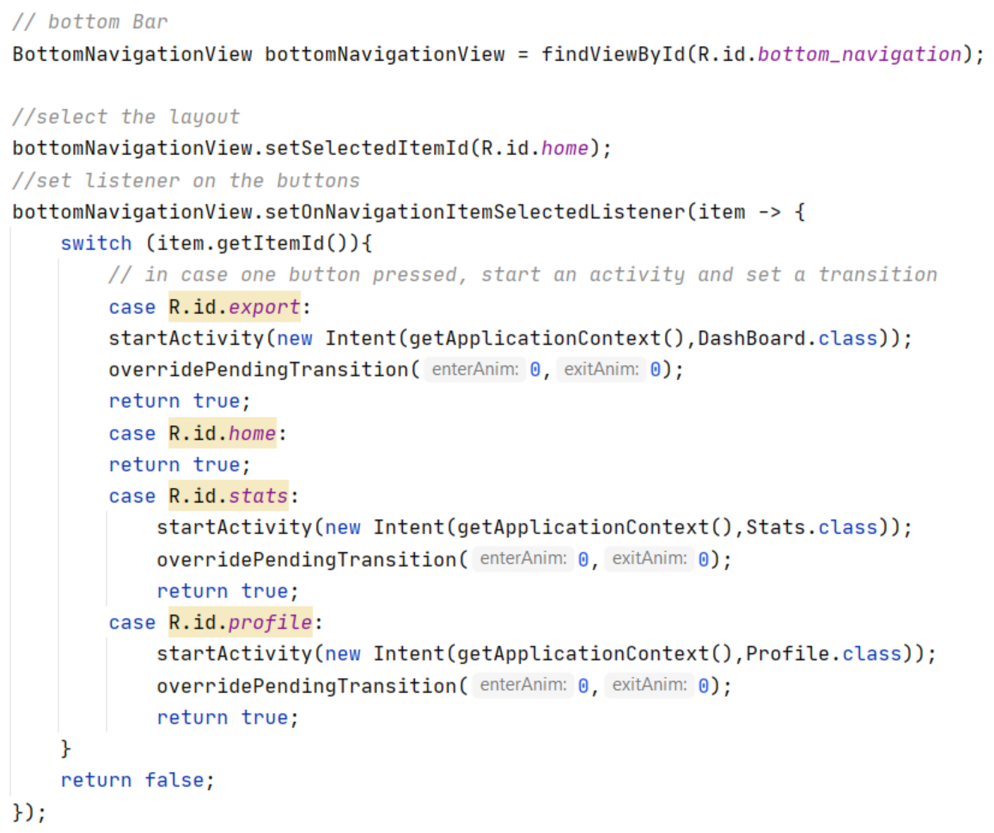
Navigation Bar

All pages have the same navigation menu, with the following options

Option	Description
Explorer	To visualize the data
Home	To return to main activity
Statistics	To plot the data



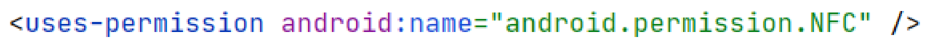
The activity code:



```java
// bottom Bar
BottomNavigationView bottomNavigationView = findViewById(R.id.bottom_navigation);

//select the layout
bottomNavigationView.setSelectedItemId(R.id.home);
//set listener on the buttons
bottomNavigationView.setOnNavigationItemSelectedListener(item -> {
    switch (item.getItemId()){
        // in case one button pressed, start an activity and set a transition
        case R.id.export:
        startActivity(new Intent(getApplicationContext(),DashBoard.class));
        overridePendingTransition( enterAnim: 0, exitAnim: 0);
        return true;
        case R.id.home:
        return true;
        case R.id.stats:
            startActivity(new Intent(getApplicationContext(),Stats.class));
            overridePendingTransition( enterAnim: 0, exitAnim: 0);
            return true;
        case R.id.profile:
            startActivity(new Intent(getApplicationContext(),Profile.class));
            overridePendingTransition( enterAnim: 0, exitAnim: 0);
            return true;
    }
    return false;
});
```

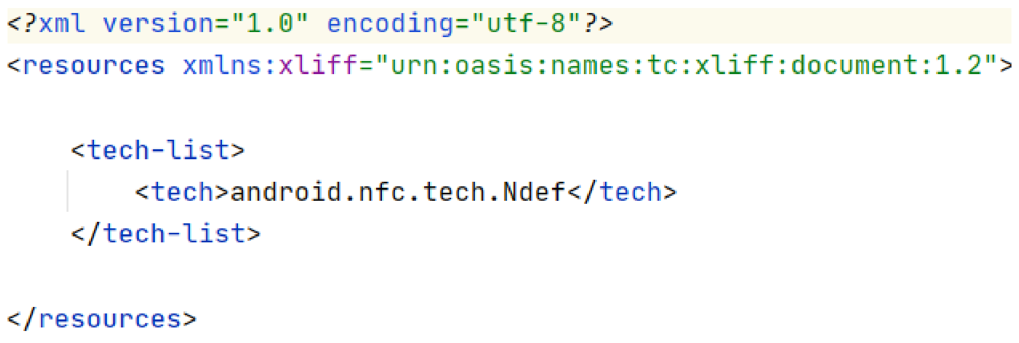
NFC

To use NFC i) permissions need to be declared in *AndroidManifest.xml*



```
<uses-permission android:name="android.permission.NFC" />
```

ii) NDEF technology needs to be declared in res/xml/nfc_tech_filter.xml

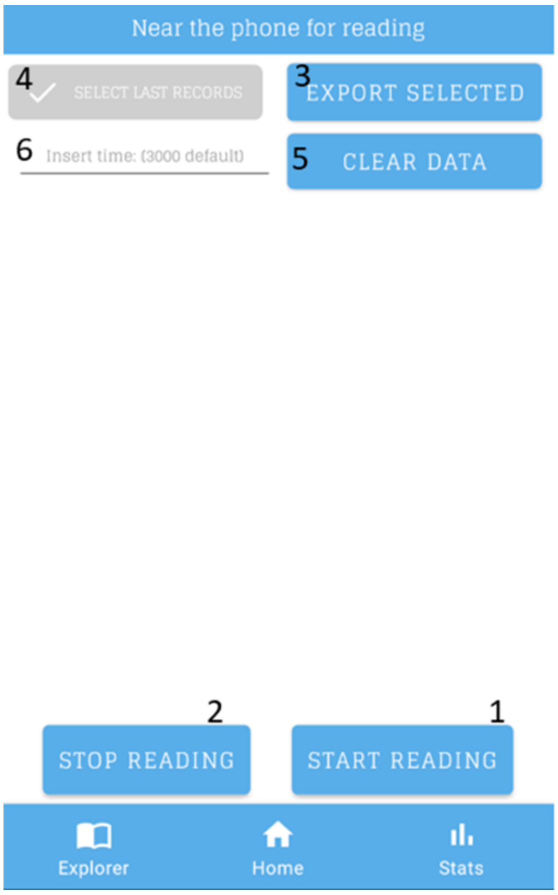


```
<?xml version="1.0" encoding="utf-8"?>
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">

    <tech-list>
        <tech>android.nfc.tech.Ndef</tech>
    </tech-list>

</resources>
```

Main Activity

At launch the application launches '*MainActivity.java*'. This is the main activity of the application; its function is to retrieve the messages from the NHS3152 chip. (Both single and multiple)

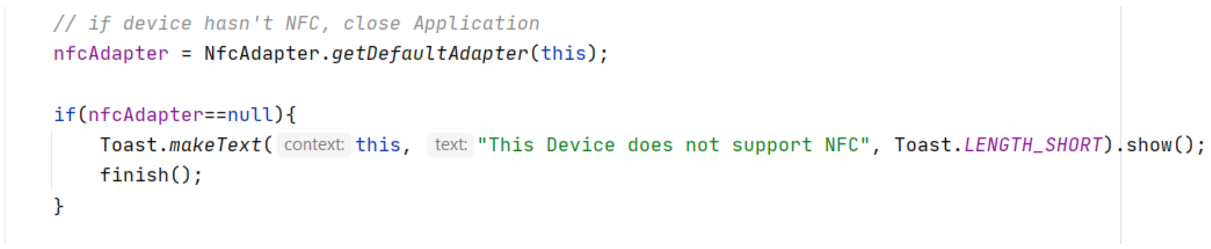
STEP	**FUCTION**
1) START READING	Start a new recording session
2) STOP READING	Stop recording session
3) EXPORTED SEsLECTED	Export selected records
4) SELECT LAST RECORDS	Select records obtain in last start-stop session
5) CLEAR DATA	Clear and eliminate on-screen data.
6) TIMED DELAY OF MESSAGE READOUT	Time between single message readouts.



Permission & specification requirement controls

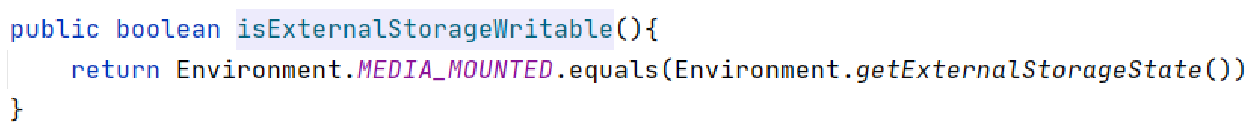
i) All graphic elements are declared.

ii) Check if Phone has NFC technology else: close application



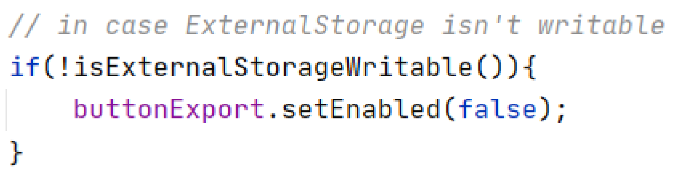
```
// if device hasn't NFC, close Application
nfcAdapter = NfcAdapter.getDefaultAdapter(this);

if(nfcAdapter==null){
    Toast.makeText( context: this,  text: "This Device does not support NFC", Toast.LENGTH_SHORT).show();
    finish();
}
```

iii) Check if we can save file in external memory



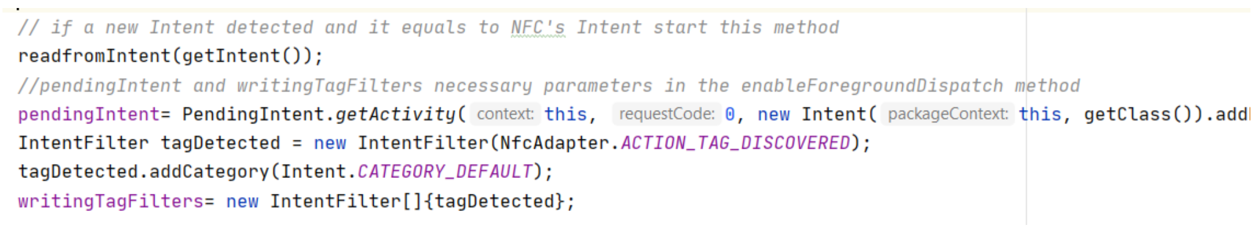
```
public boolean isExternalStorageWritable(){
    return Environment.MEDIA_MOUNTED.equals(Environment.getExternalStorageState())
}
```

If not → disable export button



```
// in case ExternalStorage isn't writable
if(!isExternalStorageWritable()){
    buttonExport.setEnabled(false);
}
```

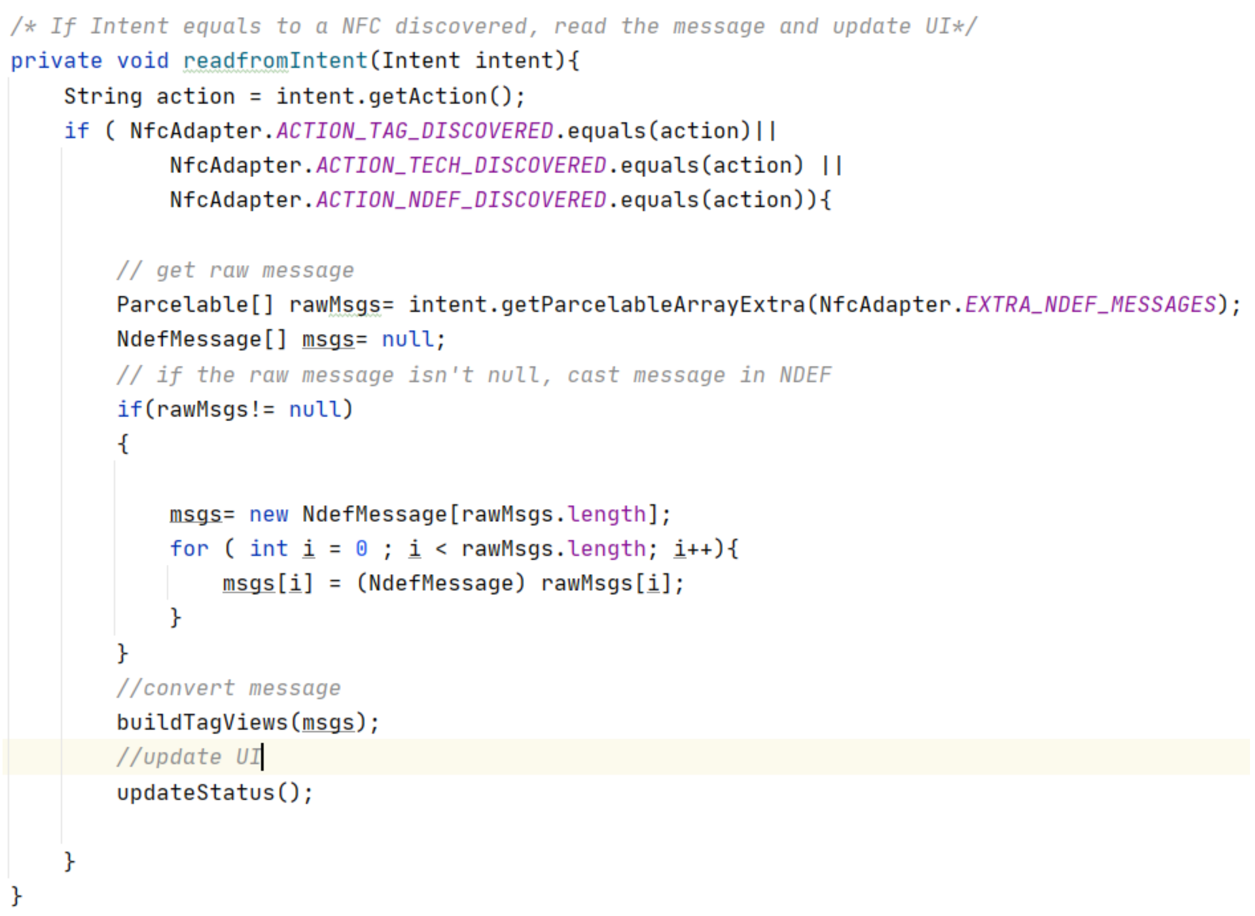
iv) Define behavior when a new intent is detected (an action, in this case the NFC tag proximity)



```
// if a new Intent detected and it equals to NFC's Intent start this method
readfromIntent(getIntent());
//pendingIntent and writingTagFilters necessary parameters in the enableForegroundDispatch method
pendingIntent= PendingIntent.getActivity( context: this,  requestCode: 0, new Intent( packageContext: this, getClass()).add
IntentFilter tagDetected = new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
tagDetected.addCategory(Intent.CATEGORY_DEFAULT);
writingTagFilters= new IntentFilter[]{tagDetected};
```

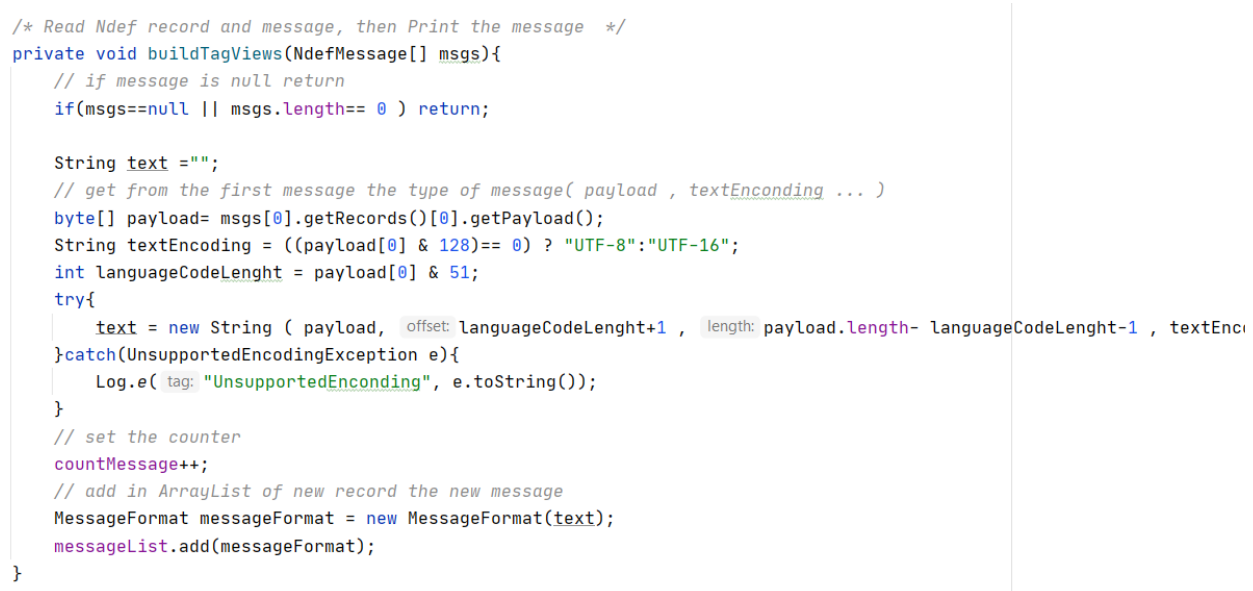
Single Message readout

i) The *readFromIntent* method controls if the new intent corresponds to NFC technology. If so → convert the received message.



```java
/* If Intent equals to a NFC discovered, read the message and update UI*/
private void readfromIntent(Intent intent){
    String action = intent.getAction();
    if ( NfcAdapter.ACTION_TAG_DISCOVERED.equals(action)||
            NfcAdapter.ACTION_TECH_DISCOVERED.equals(action) ||
            NfcAdapter.ACTION_NDEF_DISCOVERED.equals(action)){

        // get raw message
        Parcelable[] rawMsgs= intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
        NdefMessage[] msgs= null;
        // if the raw message isn't null, cast message in NDEF
        if(rawMsgs!= null)
        {

            msgs= new NdefMessage[rawMsgs.length];
            for ( int i = 0 ; i < rawMsgs.length; i++){
                msgs[i] = (NdefMessage) rawMsgs[i];
            }
        }
        //convert message
        buildTagViews(msgs);
        //update UI
        updateStatus();

    }
}
```

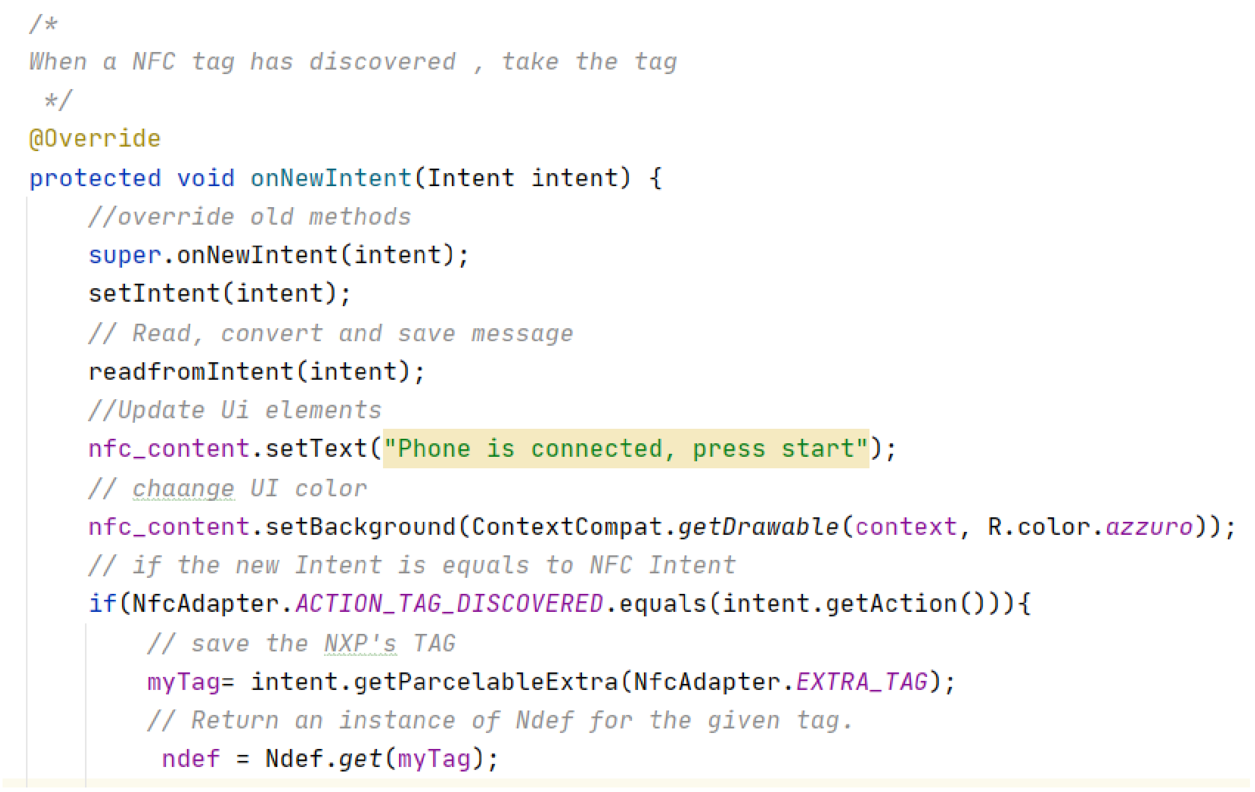
ii) the message is converted and added to the records list.



```java
/* Read Ndef record and message, then Print the message  */
private void buildTagViews(NdefMessage[] msgs){
    // if message is null return
    if(msgs==null || msgs.length== 0 ) return;

    String text ="";
    // get from the first message the type of message( payload , textEnconding ... )
    byte[] payload= msgs[0].getRecords()[0].getPayload();
    String textEncoding = ((payload[0] & 128)== 0) ? "UTF-8":"UTF-16";
    int languageCodeLenght = payload[0] & 51;
    try{
        text = new String ( payload,  offset: languageCodeLenght+1 ,  length: payload.length- languageCodeLenght-1 , textEnc
    }catch(UnsupportedEncodingException e){
        Log.e( tag: "UnsupportedEnconding", e.toString());
    }
    // set the counter
    countMessage++;
    // add in ArrayList of new record the new message
    MessageFormat messageFormat = new MessageFormat(text);
    messageList.add(messageFormat);
}
```

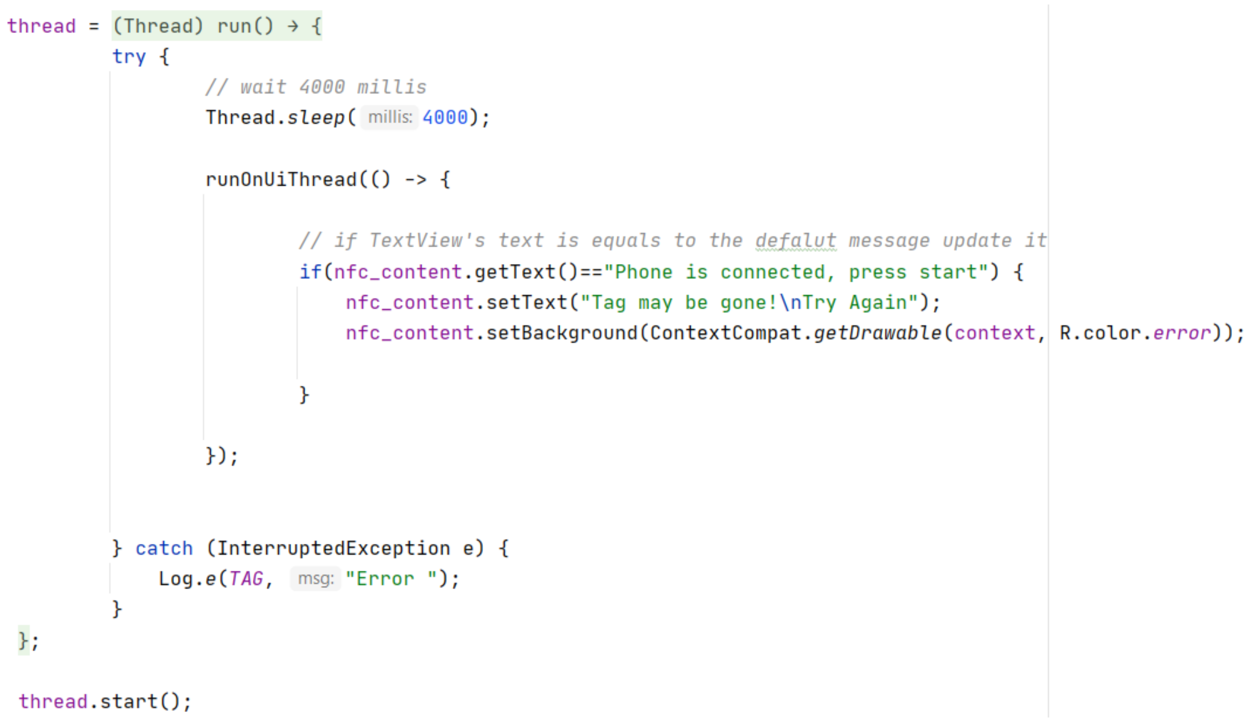
iii) The method is overwritten for the detection of a new intent (in case this didn't happen in the method *onCreate*)

iv) Detect the new message[1]



```
/*
When a NFC tag has discovered , take the tag
 */
@Override
protected void onNewIntent(Intent intent) {
    //override old methods
    super.onNewIntent(intent);
    setIntent(intent);
    // Read, convert and save message
    readfromIntent(intent);
    //Update Ui elements
    nfc_content.setText("Phone is connected, press start");
    // chaange UI color
    nfc_content.setBackground(ContextCompat.getDrawable(context, R.color.azzuro));
    // if the new Intent is equals to NFC Intent
    if(NfcAdapter.ACTION_TAG_DISCOVERED.equals(intent.getAction())){
        // save the NXP's TAG
        myTag= intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
        // Return an instance of Ndef for the given tag.
         ndef = Ndef.get(myTag);
```

v) A threat is run in which, after **4000 [ms]**, if button “start looping” is not pressed and tag is detected, an error message is printed.



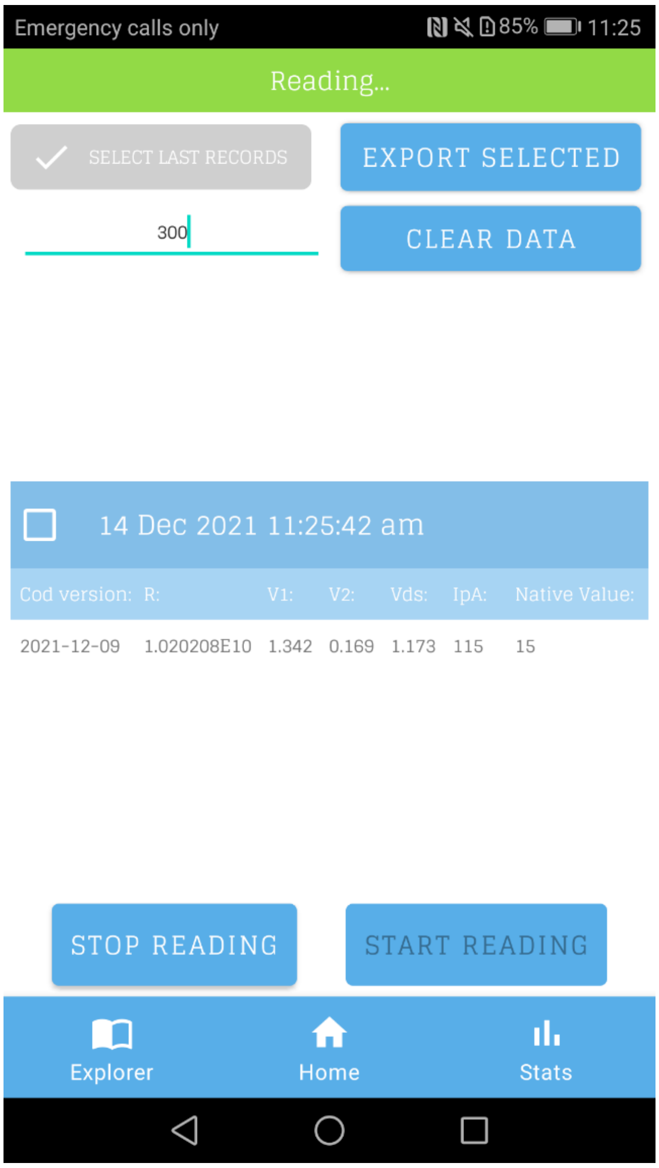
```
thread = (Thread) run() → {
        try {
                // wait 4000 millis
                Thread.sleep( millis: 4000);

                runOnUiThread(() -> {

                        // if TextView's text is equals to the defalut message update it
                        if(nfc_content.getText()=="Phone is connected, press start") {
                            nfc_content.setText("Tag may be gone!\nTry Again");
                            nfc_content.setBackground(ContextCompat.getDrawable(context, R.color.error));

                        }

                });


        } catch (InterruptedException e) {
            Log.e(TAG,  msg: "Error ");
        }
};

thread.start();
```

To start a reading in loop the phone and tag need to be connected, however it isn’t possible to control if the phone is still connected to the tag in a separate thread. If tag is not connected any longer, it crashes the

[1] This part may extract the RFID unique ID – but we aren’t certain. – check if you need Unique ID.

device. Thus, the problem is solved by disabling loop reading after a certain amount of time, requiring it to connect to tag again

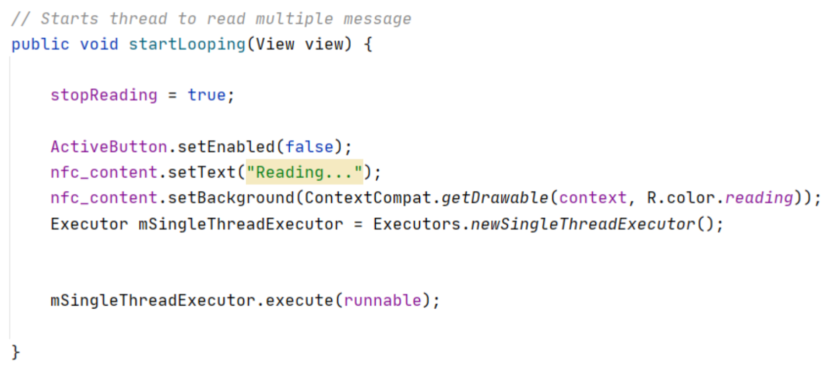
Multiple Messages readout



14 Dec 2021 11:25:42 am

Cod version:	R:	V1:	V2:	Vds:	IpA:	Native Value:
2021-12-09	1.020208E10	1.342	0.169	1.173	115	15

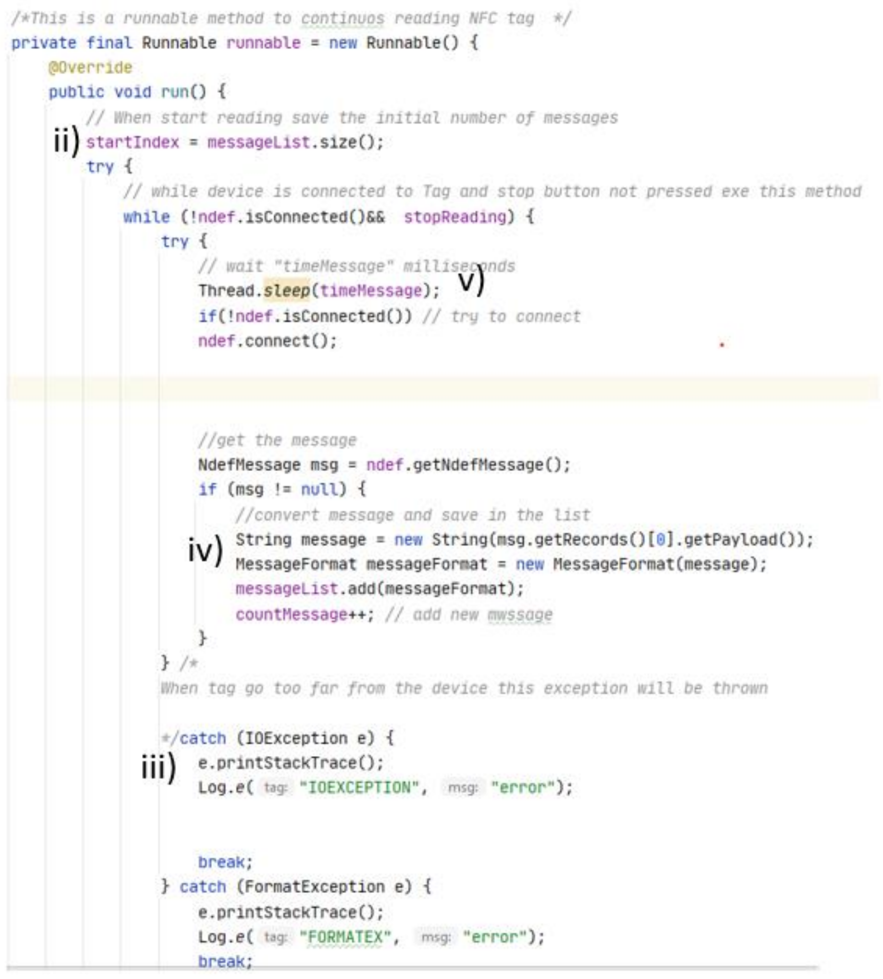
When Start Reading is pressed, (with phone-tag connected) the following method is launched, which **i)** starts the Multiple Messages readout thread.



```
// Starts thread to read multiple message
public void startLooping(View view) {

    stopReading = true;

    ActiveButton.setEnabled(false);
    nfc_content.setText("Reading...");
    nfc_content.setBackground(ContextCompat.getDrawable(context, R.color.reading));
    Executor mSingleThreadExecutor = Executors.newSingleThreadExecutor();


    mSingleThreadExecutor.execute(runnable);

}
```

ii) Saves the index (the array dimensions) in which reading starts. This is needed for future export options.

The reading continues until while these 3 conditions are met: The phone tag are connected, *STOP* button isn't pressed, **iii)** The phone tag are within range. **iv)** the message is received, converted and saved.



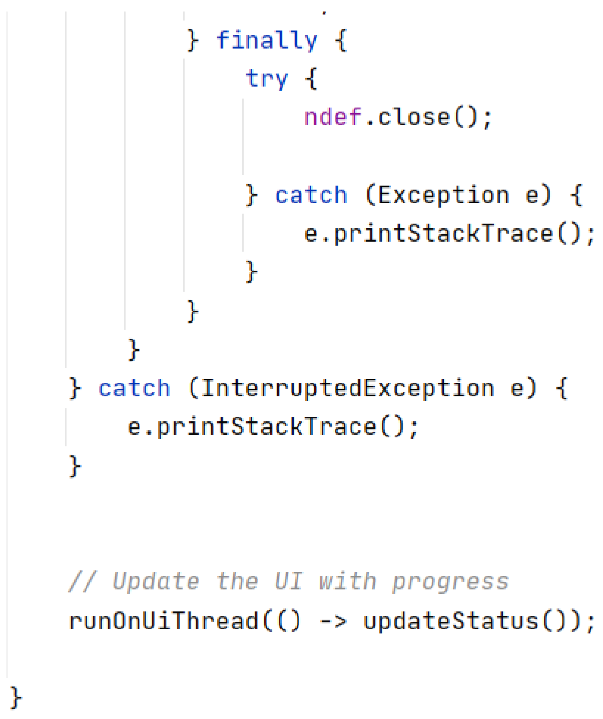
```
/*This is a runnable method to continuos reading NFC tag  */
private final Runnable runnable = new Runnable() {
    @Override
    public void run() {
        // When start reading save the initial number of messages
        startIndex = messageList.size();
        try {
            // while device is connected to Tag and stop button not pressed exe this method
            while (!ndef.isConnected()&&  stopReading) {
                try {
                    // wait "timeMessage" milliseconds
                    Thread.sleep(timeMessage);
                    if(!ndef.isConnected()) // try to connect
                    ndef.connect();



                    //get the message
                    NdefMessage msg = ndef.getNdefMessage();
                    if (msg != null) {
                        //convert message and save in the list
                        String message = new String(msg.getRecords()[0].getPayload());
                        MessageFormat messageFormat = new MessageFormat(message);
                        messageList.add(messageFormat);
                        countMessage++; // add new mwssage
                    }
                } /*
                When tag go too far from the device this exception will be thrown

                */catch (IOException e) {
                    e.printStackTrace();
                    Log.e( tag: "IOEXCEPTION",  msg: "error");


                    break;
                } catch (FormatException e) {
                    e.printStackTrace();
                    Log.e( tag: "FORMATEX",  msg: "error");
                    break;
```

Delay time for message reception is given by variable v) *timeMessage* which (if specified) gets the value in the *EditBox*. (**default 3000 [ms]**)

Closing the connection

If at least one of the 3 conditions above are met, the connection is closed with the command *close*(), and the graphic elements are updated.

Since it is a threat different from the Main Thread, to update the UI, the method *updateStatus()* is called from *runOnUiThread*.

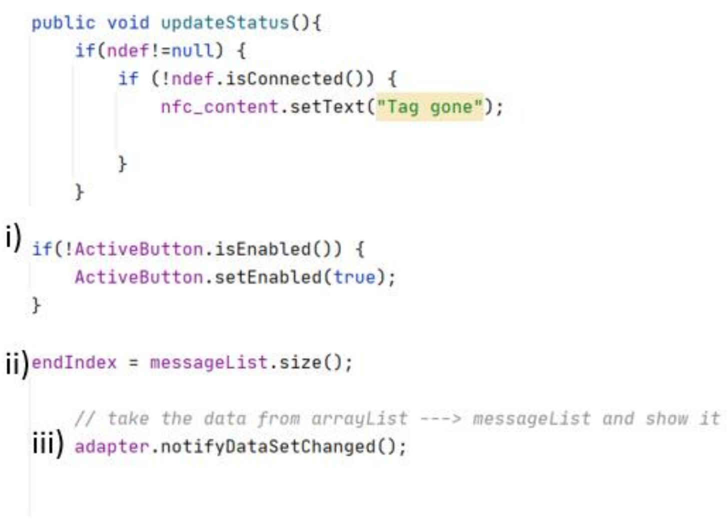


```
            } finally {
                try {
                    ndef.close();

                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }


    // Update the UI with progress
    runOnUiThread(() -> updateStatus());

}
```

Update Status

This method is called to update the *RecyclerView* containing the different measurements.

The method updates the button i) Start Reading, ii) assigns to the variable that identifies the end of the measurements the value of the *arrayList* after measurement is done, and iii) notifies to RecyclerView that the data is changed.

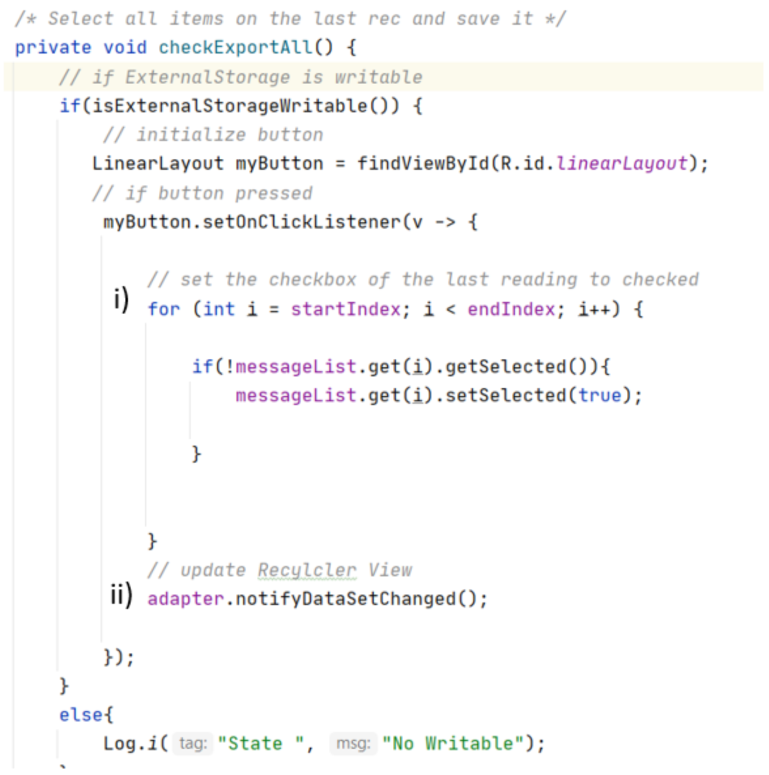


```
public void updateStatus(){
    if(ndef!=null) {
        if (!ndef.isConnected()) {
            nfc_content.setText("Tag gone");

        }
    }
i) if(!ActiveButton.isEnabled()) {
    ActiveButton.setEnabled(true);
}

ii)endIndex = messageList.size();

    // take the data from arrayList ---> messageList and show it
iii) adapter.notifyDataSetChanged();
```

Export Options

Selection of last measurement

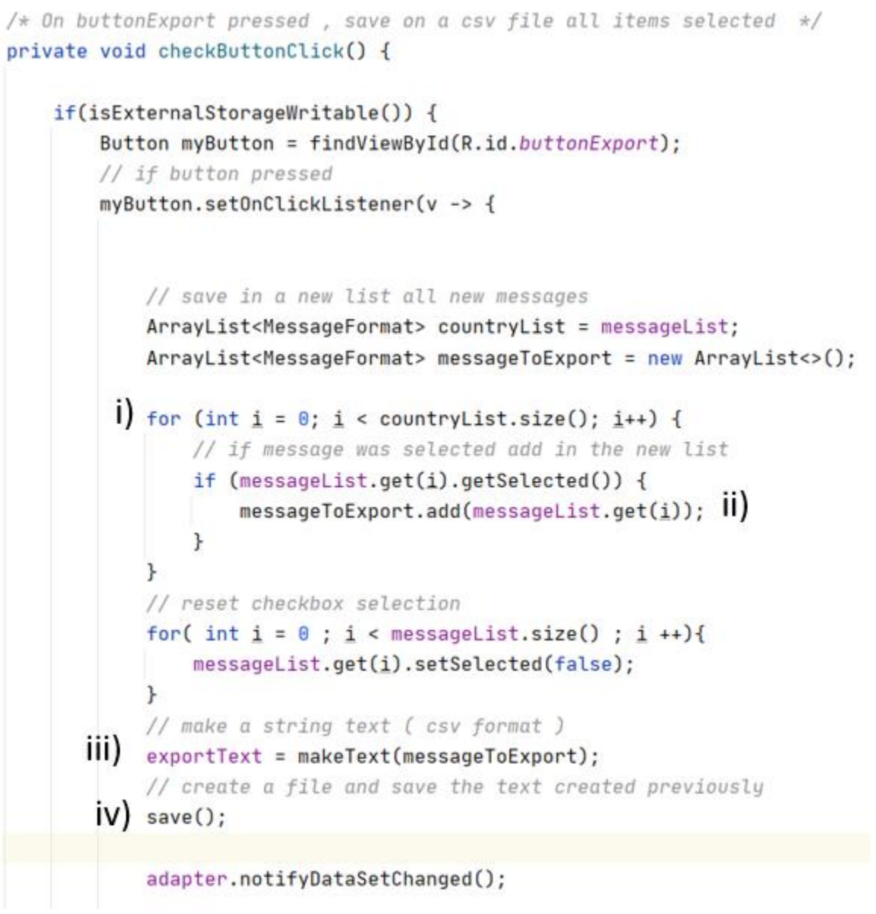
This method controls **i)** the values of the indexes from the start of the last measurement, until the end of it, **ii)** selects all the *checkboxes* that are within this range and notifies *RecyclerView*.



```
/* Select all items on the last rec and save it */
private void checkExportAll() {
    // if ExternalStorage is writable
    if(isExternalStorageWritable()) {
        // initialize button
       LinearLayout myButton = findViewById(R.id.linearLayout);
       // if button pressed
        myButton.setOnClickListener(v -> {

     i)     // set the checkbox of the last reading to checked
            for (int i = startIndex; i < endIndex; i++) {

                if(!messageList.get(i).getSelected()){
                    messageList.get(i).setSelected(true);

                }


            }
            // update Recylcler View
     ii)    adapter.notifyDataSetChanged();

        });
    }
    else{
        Log.i( tag: "State ",  msg: "No Writable");
```

Export Selected

This method **i)** controls that for each measurement, if the checkbox has been selected, **ii)** adds the measurements to a new *ArrayList*, **iii)** mergers each measurement (*makeText(),* and **iv)** saves each file as a *.csv (save(*)).



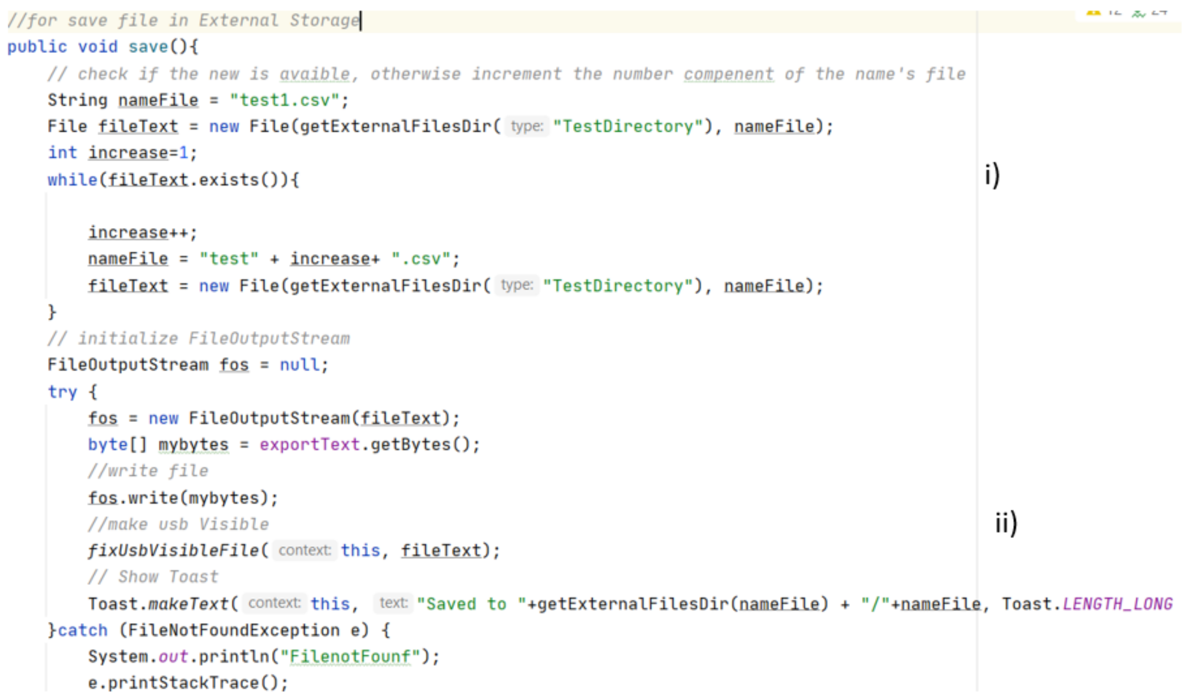
```java
/* On buttonExport pressed , save on a csv file all items selected  */
private void checkButtonClick() {

    if(isExternalStorageWritable()) {
        Button myButton = findViewById(R.id.buttonExport);
        // if button pressed
        myButton.setOnClickListener(v -> {


            // save in a new list all new messages
            ArrayList<MessageFormat> countryList = messageList;
            ArrayList<MessageFormat> messageToExport = new ArrayList<>();

            for (int i = 0; i < countryList.size(); i++) {
                // if message was selected add in the new list
                if (messageList.get(i).getSelected()) {
                    messageToExport.add(messageList.get(i));
                }
            }
            // reset checkbox selection
            for( int i = 0 ; i < messageList.size() ; i ++){
                messageList.get(i).setSelected(false);
            }
            // make a string text ( csv format )
            exportText = makeText(messageToExport);
            // create a file and save the text created previously
            save();

            adapter.notifyDataSetChanged();
```

Save()

All files are saved in *Android/Data/com.example.readerNHS/TestDirectory/*

i) Check if the name of the file already exists, if so → increment the value until you find a new filename not in directory.

ii) The file is composed, from the text string extracted previously. The file is saved in the external memory (if mounted) with the method *fixUsbVisibleFile().*

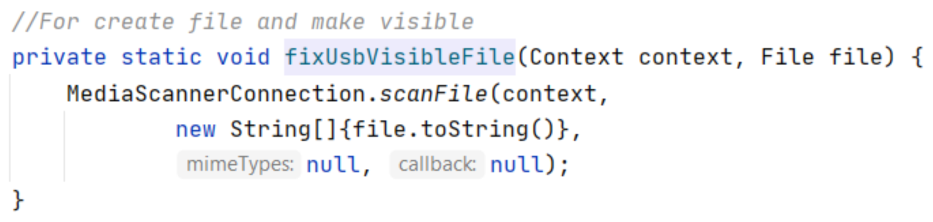


```
//for save file in External Storage
public void save(){
    // check if the new is avaible, otherwise increment the number compenent of the name's file
    String nameFile = "test1.csv";
    File fileText = new File(getExternalFilesDir( type: "TestDirectory"), nameFile);
    int increase=1;
    while(fileText.exists()){

        increase++;
        nameFile = "test" + increase+ ".csv";
        fileText = new File(getExternalFilesDir( type: "TestDirectory"), nameFile);
    }
    // initialize FileOutputStream
    FileOutputStream fos = null;
    try {
        fos = new FileOutputStream(fileText);
        byte[] mybytes = exportText.getBytes();
        //write file
        fos.write(mybytes);
        //make usb Visible
        fixUsbVisibleFile( context: this, fileText);
        // Show Toast
        Toast.makeText( context: this,  text: "Saved to "+getExternalFilesDir(nameFile) + "/"+nameFile, Toast.LENGTH_LONG
    }catch (FileNotFoundException e) {
        System.out.println("FilenotFounf");
        e.printStackTrace();
```

i)

ii)

This method used the library *MediaScanner* to save the file and make it visible even to the user. (Even without root mode).

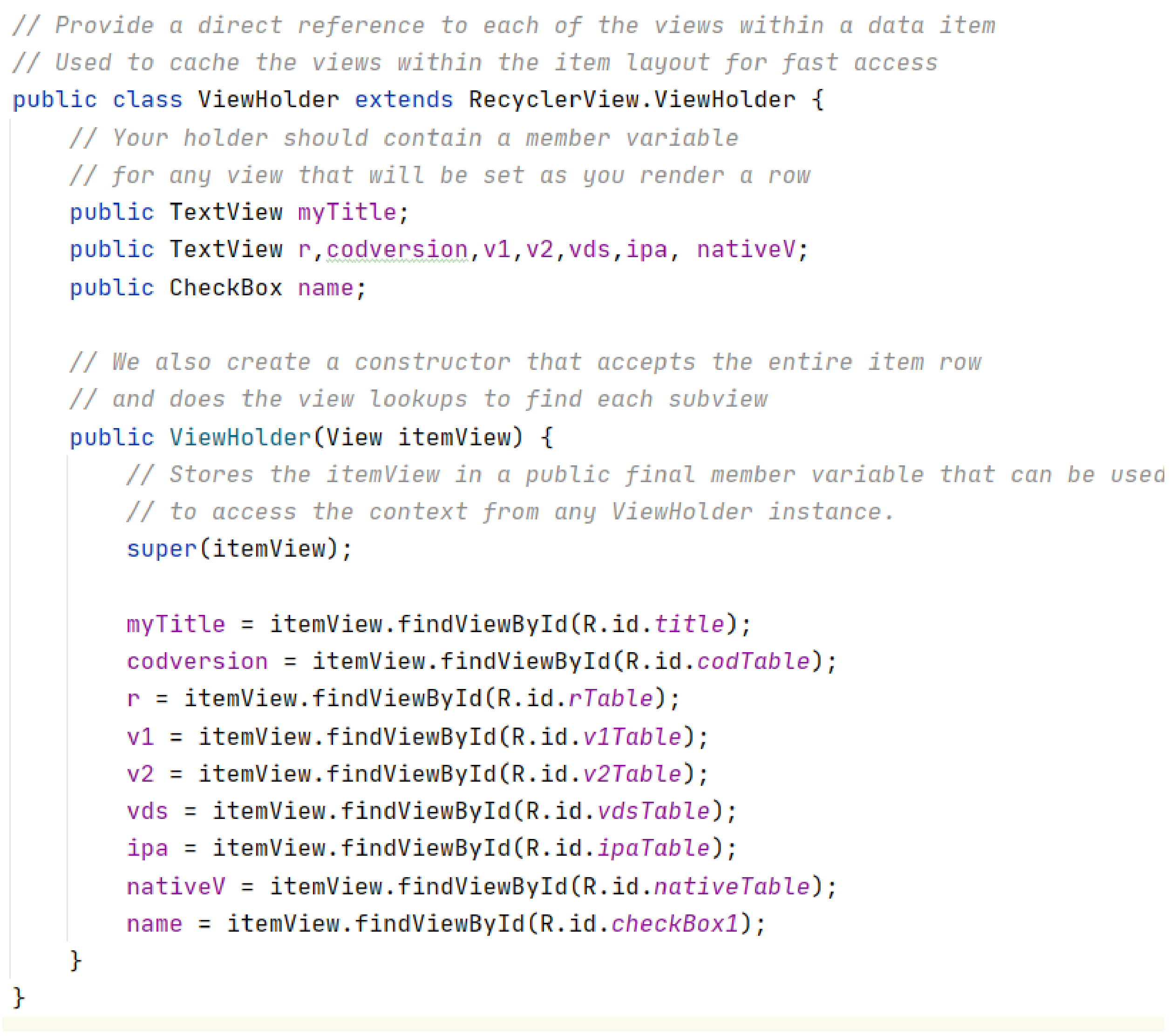


```
//For create file and make visible
private static void fixUsbVisibleFile(Context context, File file) {
    MediaScannerConnection.scanFile(context,
            new String[]{file.toString()},
             mimeTypes: null,  callback: null);
}
```

Recycler View

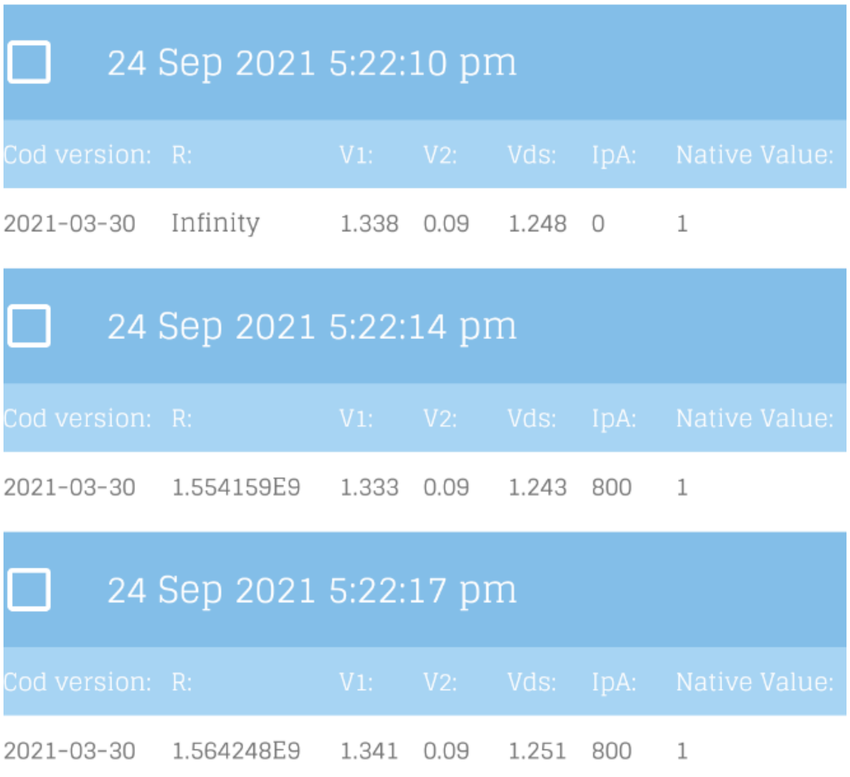
To visualize the different measurements, one for each line, *recycleview* is used. This is the same one used in page explorer. A specific Adapter ("*RecordAdapter*") is used on the bases of the ones available from *AndoridStudio*.

To visualize the different records, we define a *ViewHolder*:



```java
// Provide a direct reference to each of the views within a data item
// Used to cache the views within the item layout for fast access
public class ViewHolder extends RecyclerView.ViewHolder {
    // Your holder should contain a member variable
    // for any view that will be set as you render a row
    public TextView myTitle;
    public TextView r,codversion,v1,v2,vds,ipa, nativeV;
    public CheckBox name;

    // We also create a constructor that accepts the entire item row
    // and does the view lookups to find each subview
    public ViewHolder(View itemView) {
        // Stores the itemView in a public final member variable that can be used
        // to access the context from any ViewHolder instance.
        super(itemView);

        myTitle = itemView.findViewById(R.id.title);
        codversion = itemView.findViewById(R.id.codTable);
        r = itemView.findViewById(R.id.rTable);
        v1 = itemView.findViewById(R.id.v1Table);
        v2 = itemView.findViewById(R.id.v2Table);
        vds = itemView.findViewById(R.id.vdsTable);
        ipa = itemView.findViewById(R.id.ipaTable);
        nativeV = itemView.findViewById(R.id.nativeTable);
        name = itemView.findViewById(R.id.checkBox1);
    }
}
```

The result:



24 Sep 2021 5:22:10 pm

Cod version:	R:	V1:	V2:	Vds:	IpA:	Native Value:
2021-03-30	Infinity	1.338	0.09	1.248	0	1

24 Sep 2021 5:22:14 pm

Cod version:	R:	V1:	V2:	Vds:	IpA:	Native Value:
2021-03-30	1.554159E9	1.333	0.09	1.243	800	1

24 Sep 2021 5:22:17 pm

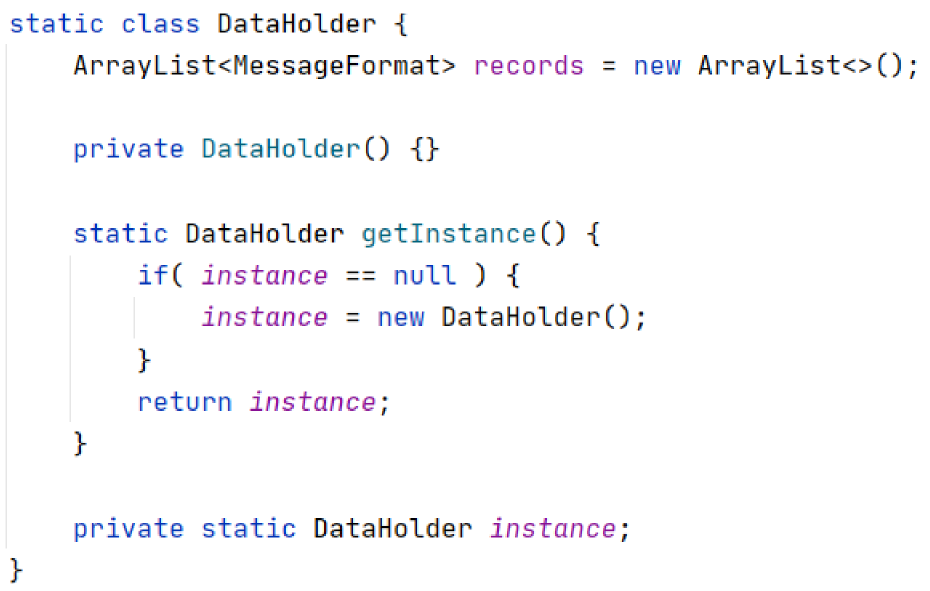
Cod version:	R:	V1:	V2:	Vds:	IpA:	Native Value:
2021-03-30	1.564248E9	1.341	0.09	1.251	800	1

To be able to select the records, in the adaptor method*, onBindViewHolder()* is defined. This is a Checkbox listener that acts directly with the attribute of the record list.

```
//nel caso venga premuta la checkbox viene settato l'attributo del messaggio come Selected
holder.name.setOnClickListener(
        v -> {
            CheckBox cb = (CheckBox) v;

            messageList.get(position).setSelected(cb.isChecked());
        });
```

Data persistence and correct use of NFC technology

To maintain the data in case the application is paused, or activity changed, we use *DataHolder*

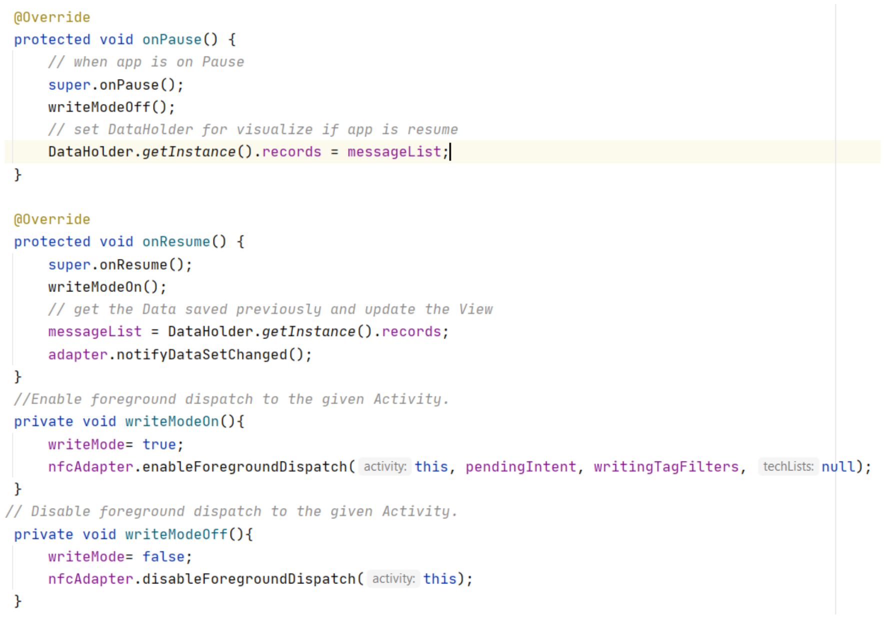


```
static class DataHolder {
    ArrayList<MessageFormat> records = new ArrayList<>();

    private DataHolder() {}

    static DataHolder getInstance() {
        if( instance == null ) {
            instance = new DataHolder();
        }
        return instance;
    }

    private static DataHolder instance;
}
```

DataHolder is called in the methods *OnResume()* and *OnPause(),* together with *writeModeOn()* and *writeModeOff(),* which are method to enable to use the technology while changing from running to pause.[2]

Also, when the application passes from the active state to pause, the *"Foreground Dispatcher"* is disabled. I fit passed from "*pause*" to "Resume", the "Foreground Dispatcher" is enabled.

This is the routine necessary for NFC technology use:



```
@Override
protected void onPause() {
    // when app is on Pause
    super.onPause();
    writeModeOff();
    // set DataHolder for visualize if app is resume
    DataHolder.getInstance().records = messageList;
}

@Override
protected void onResume() {
    super.onResume();
    writeModeOn();
    // get the Data saved previously and update the View
    messageList = DataHolder.getInstance().records;
    adapter.notifyDataSetChanged();
}
//Enable foreground dispatch to the given Activity.
private void writeModeOn(){
    writeMode= true;
    nfcAdapter.enableForegroundDispatch( activity: this, pendingIntent, writingTagFilters,  techLists: null);
}
// Disable foreground dispatch to the given Activity.
private void writeModeOff(){
    writeMode= false;
    nfcAdapter.disableForegroundDispatch( activity: this);
}
```

[2] Don't know how it works, but it works – don't fix.

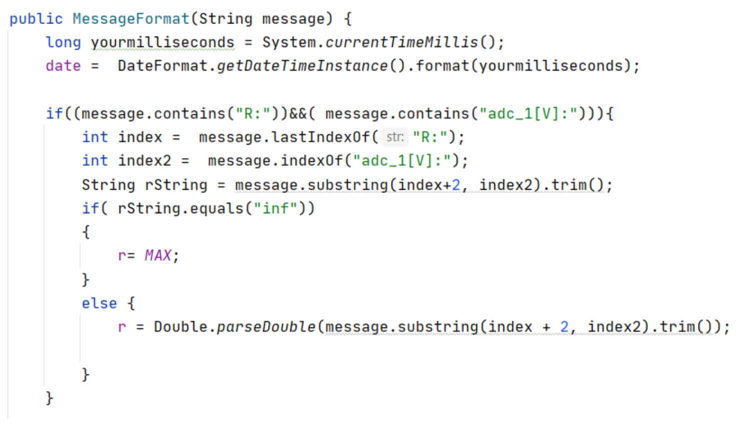
Messageformat

This class defines the format of the messages. It extracts the values from the converted message (from string), the message needs to be of a pre-established type, and in case the message changes, this will be the class to modify, together with *makeText()* in Export of *MainActivity()*.

The class defines 2 constructions

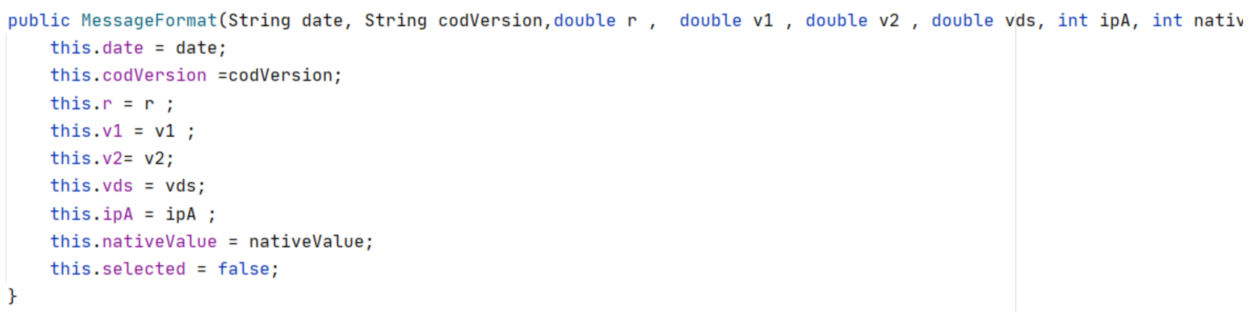
1. The first in which i) we pass as parameter the string to the message, ii) through the methods that contains substrings it extracts the values, iii) assigns those values to an instance of class *MessageFormat*.

This is one part of the code, for the others the logic is similar:



```
public MessageFormat(String message) {
    long yourmilliseconds = System.currentTimeMillis();
    date =  DateFormat.getDateTimeInstance().format(yourmilliseconds);

    if((message.contains("R:"))&&( message.contains("adc_1[V]:"))){
        int index =  message.lastIndexOf( str: "R:");
        int index2 =  message.indexOf("adc_1[V]:");
        String rString = message.substring(index+2, index2).trim();
        if( rString.equals("inf"))
        {
            r= MAX;
        }
        else {
            r = Double.parseDouble(message.substring(index + 2, index2).trim());

        }
    }
```

2. A second constructor to create an instance of *MessageFormat* having already the values of the measurement:



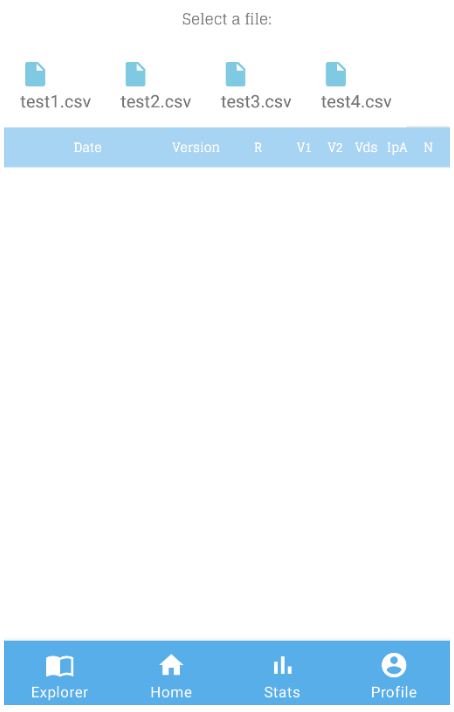
```
public MessageFormat(String date, String codVersion,double r ,  double v1 , double v2 , double vds, int ipA, int nativ
    this.date = date;
    this.codVersion =codVersion;
    this.r = r ;
    this.v1 = v1 ;
    this.v2= v2;
    this.vds = vds;
    this.ipA = ipA ;
    this.nativeValue = nativeValue;
    this.selected = false;
}
```

3.

Dashboard

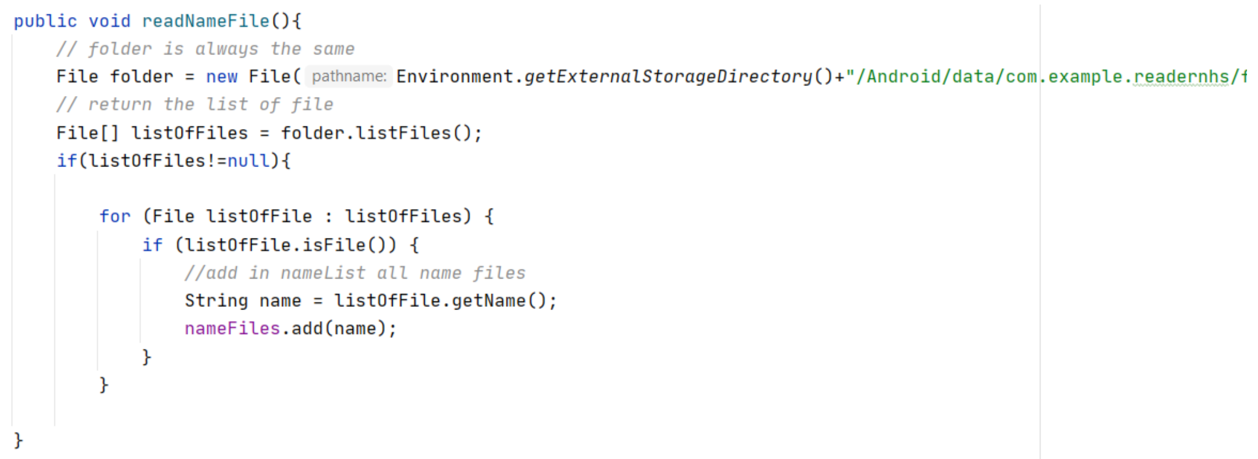
In this Activity it is possible to read the files saved previously, pressing on interested file the list of records will open.

The activity is composed from 2 *RecycleView*, one the same as for *MainActivity*, the other for the visualization of names of the files, positioned top of the screen, horizontally.



After having declared and initialized all the graphic elements of the *Activity*, the method *readNameFile()* is run.

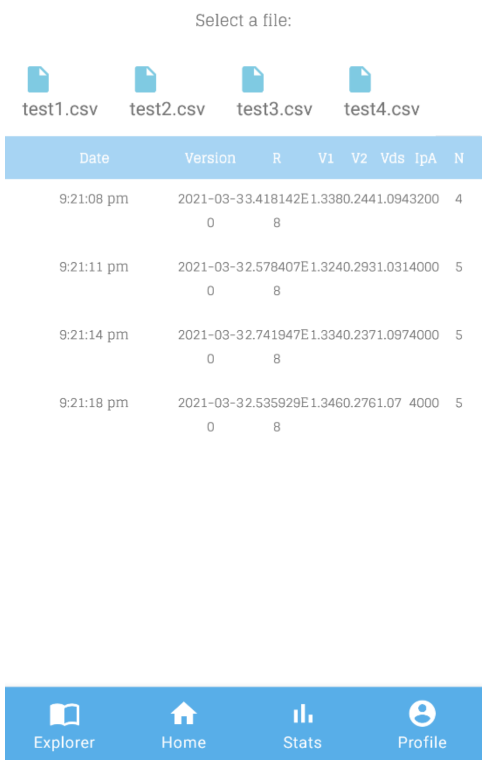
This method takes the folder (already set, **FOR FUTURE CHANGES MODIFY THIS LINE**) and saves in *ArrayList* the different file names.



```
public void readNameFile(){
    // folder is always the same
    File folder = new File( pathname: Environment.getExternalStorageDirectory()+"/Android/data/com.example.readernhs/f
    // return the list of file
    File[] listOfFiles = folder.listFiles();
    if(listOfFiles!=null){

        for (File listOfFile : listOfFiles) {
            if (listOfFile.isFile()) {
                //add in nameList all name files
                String name = listOfFile.getName();
                nameFiles.add(name);
            }
        }

}
```

RecyclerView is notified of the data changes.

In the case that the name or the figure of the file is pressed, the method for the reading and visualization of the records in initialized *readFIles()*. Here is an example after having pressed one of the files:

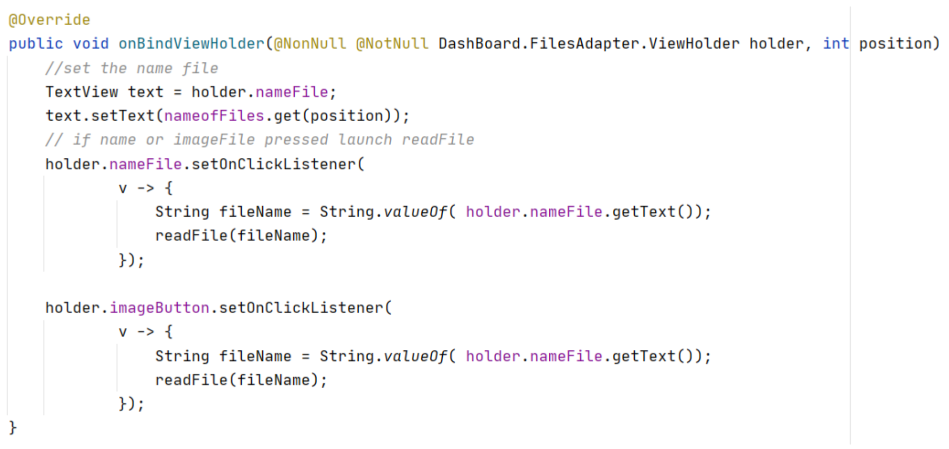


Select a file:

test1.csv test2.csv test3.csv test4.csv

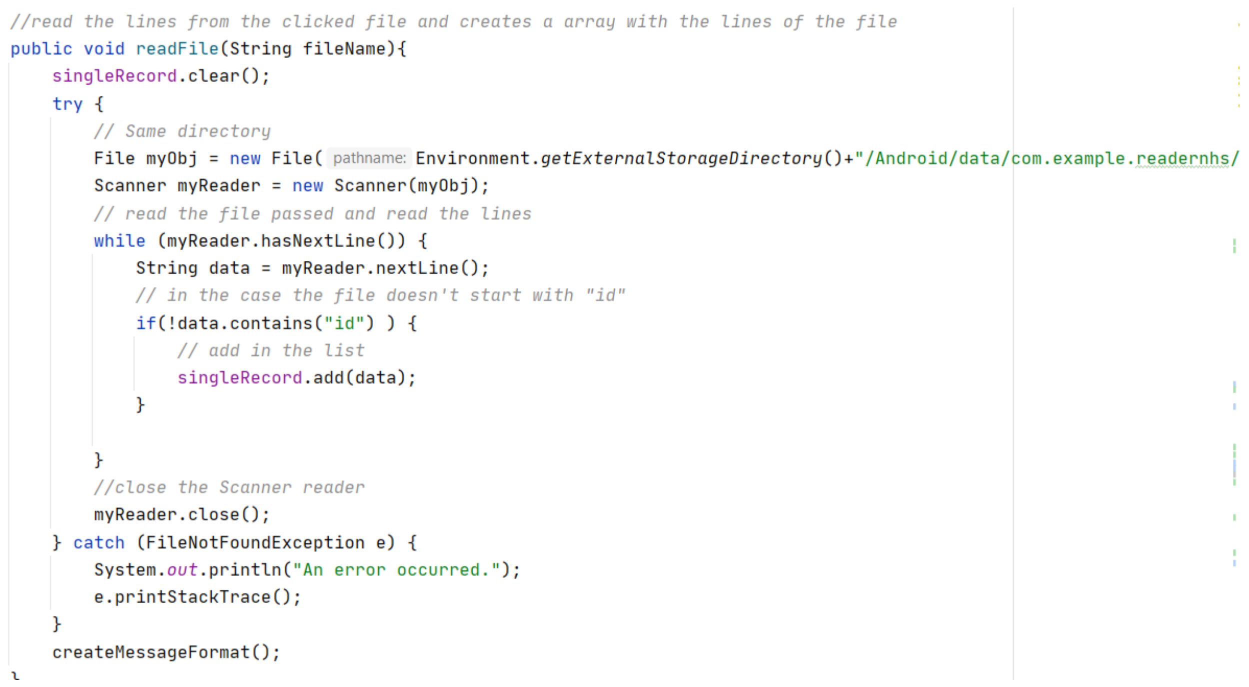
Date	Version	R	V1	V2	Vds	IpA	N
9:21:08 pm	2021-03-30	3.418142E8	1.338	0.244	1.094	3200	4
9:21:11 pm	2021-03-30	2.578407E8	1.324	0.293	1.031	4000	5
9:21:14 pm	2021-03-30	2.741947E8	1.334	0.237	1.097	4000	5
9:21:18 pm	2021-03-30	2.535929E8	1.346	0.276	1.07	4000	5

i) This method is a part of the *Adapter* code, which describes the behavior in the visualization and in the case that an element of the view is pressed:



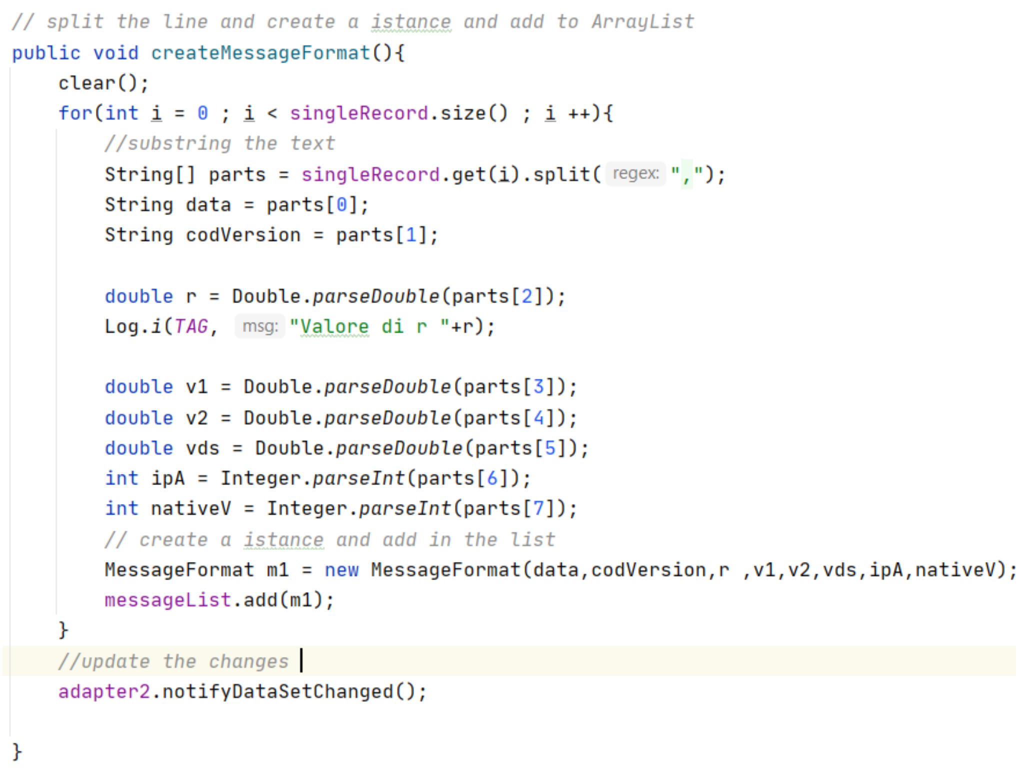
```
@Override
public void onBindViewHolder(@NonNull @NotNull DashBoard.FilesAdapter.ViewHolder holder, int position)
    //set the name file
    TextView text = holder.nameFile;
    text.setText(nameofFiles.get(position));
    // if name or imageFile pressed launch readFile
    holder.nameFile.setOnClickListener(
            v -> {
                String fileName = String.valueOf( holder.nameFile.getText());
                readFile(fileName);
            });

    holder.imageButton.setOnClickListener(
            v -> {
                String fileName = String.valueOf( holder.nameFile.getText());
                readFile(fileName);
            });
}
```

ii) this method takes the filename as parameter (of pressed file), extracts all lines, and saves it to an *ArrayList*:



```
//read the lines from the clicked file and creates a array with the lines of the file
public void readFile(String fileName){
    singleRecord.clear();
    try {
        // Same directory
        File myObj = new File( pathname: Environment.getExternalStorageDirectory()+"/Android/data/com.example.readernhs/
        Scanner myReader = new Scanner(myObj);
        // read the file passed and read the lines
        while (myReader.hasNextLine()) {
            String data = myReader.nextLine();
            // in the case the file doesn't start with "id"
            if(!data.contains("id") ) {
                // add in the list
                singleRecord.add(data);
            }

        }
        //close the Scanner reader
        myReader.close();
    } catch (FileNotFoundException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
    createMessageFormat();
}
```

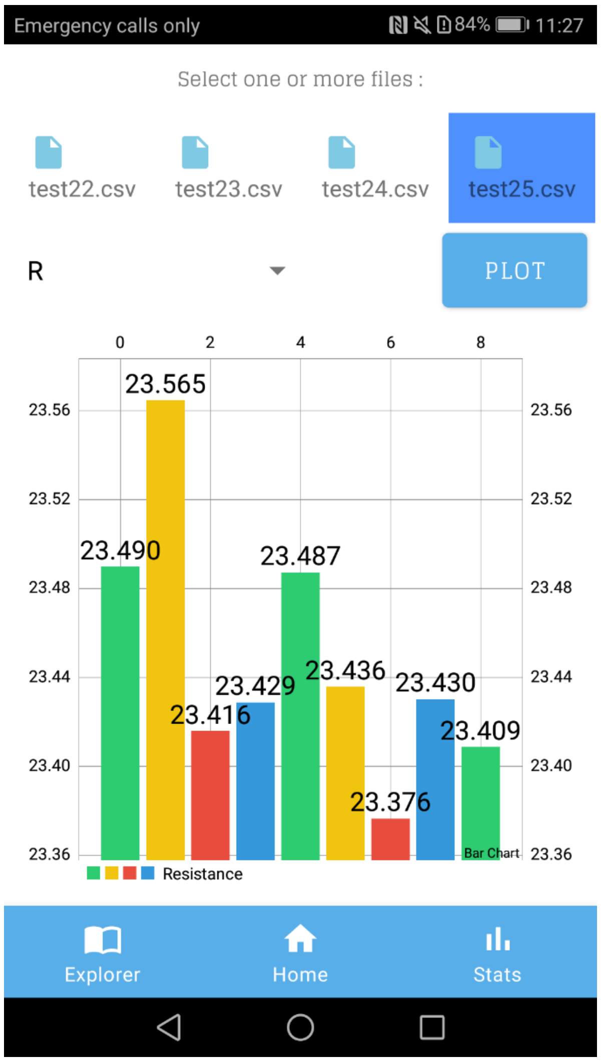
iii) we execute the *CreateMessageFormat*() that splits the different lines extracted from the files, and the visualization in the appropriate View.



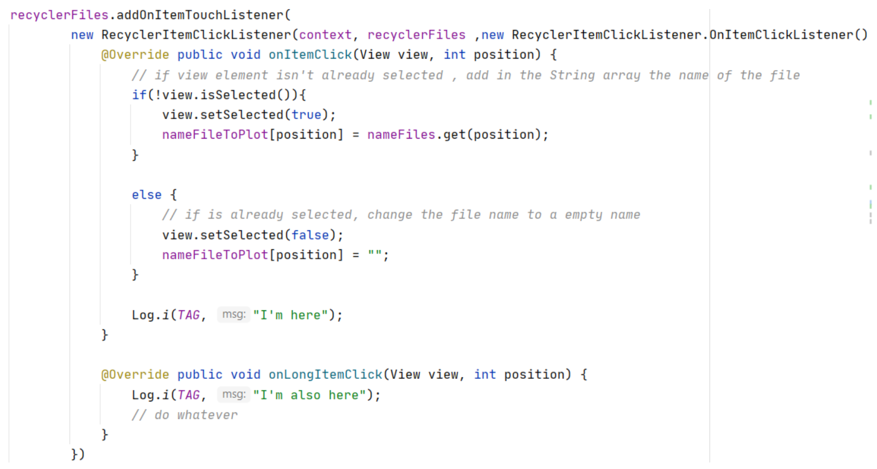
```
// split the line and create a istance and add to ArrayList
public void createMessageFormat(){
    clear();
    for(int i = 0 ; i < singleRecord.size() ; i ++){
        //substring the text
        String[] parts = singleRecord.get(i).split( regex: ",");
        String data = parts[0];
        String codVersion = parts[1];

        double r = Double.parseDouble(parts[2]);
        Log.i(TAG,  msg: "Valore di r "+r);

        double v1 = Double.parseDouble(parts[3]);
        double v2 = Double.parseDouble(parts[4]);
        double vds = Double.parseDouble(parts[5]);
        int ipA = Integer.parseInt(parts[6]);
        int nativeV = Integer.parseInt(parts[7]);
        // create a istance and add in the list
        MessageFormat m1 = new MessageFormat(data,codVersion,r ,v1,v2,vds,ipA,nativeV);
        messageList.add(m1);
    }
    //update the changes
    adapter2.notifyDataSetChanged();

}
```

Statistics

This Activity it is possible to select one of more files (through a *Recycleview* similar to the one in dashboard) and display a graph with the different records contained in the selected files. This is made possible using the pre-defined class *RecyclerItemClickListener* that allows to remain listening for click to different elements in *Recyclerview*.



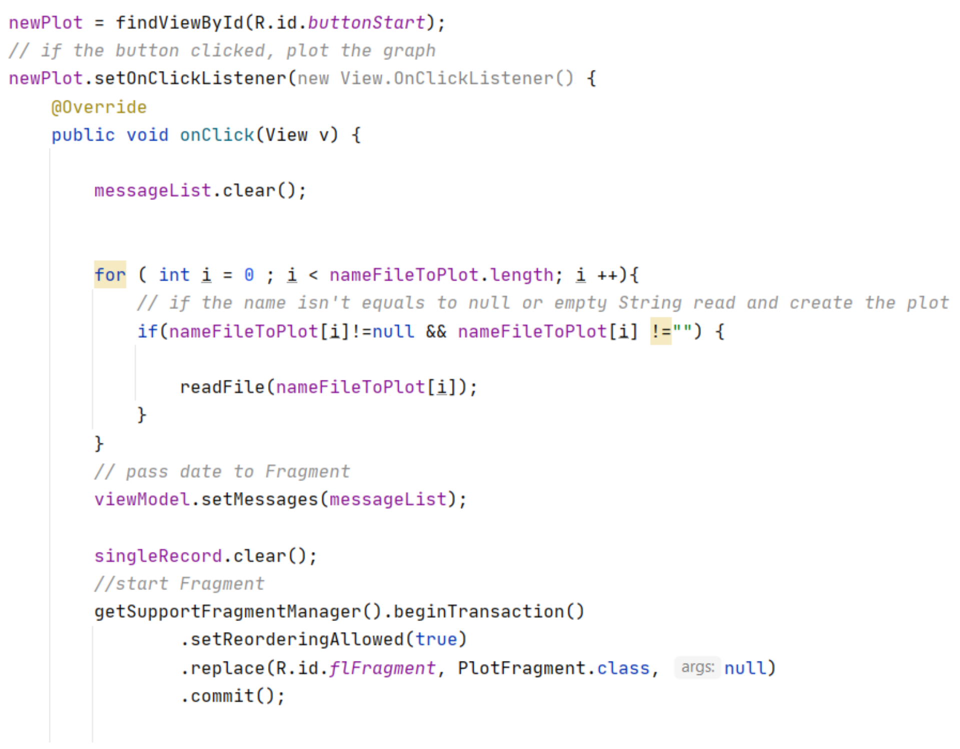
When an element in *Recycler* is touched, this is selected and highlighted:



```
recyclerFiles.addOnItemTouchListener(
        new RecyclerItemClickListener(context, recyclerFiles ,new RecyclerItemClickListener.OnItemClickListener()
            @Override public void onItemClick(View view, int position) {
                // if view element isn't already selected , add in the String array the name of the file
                if(!view.isSelected()){
                    view.setSelected(true);
                    nameFileToPlot[position] = nameFiles.get(position);
                }

                else {
                    // if is already selected, change the file name to a empty name
                    view.setSelected(false);
                    nameFileToPlot[position] = "";
                }

                Log.i(TAG,  msg: "I'm here");
            }

            @Override public void onLongItemClick(View view, int position) {
                Log.i(TAG,  msg: "I'm also here");
                // do whatever
            }
        })
```

This is made possible thanks to the Class *RecyclerItemClickListener* available form Android.

When the PLOT button is pressed, the following method is initialized in which the data is extracted, and the graphic *Fragment* is started passing the values through a *SharedModel*:



```
newPlot = findViewById(R.id.buttonStart);
// if the button clicked, plot the graph
newPlot.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {

        messageList.clear();


        for ( int i = 0 ; i < nameFileToPlot.length; i ++){
            // if the name isn't equals to null or empty String read and create the plot
            if(nameFileToPlot[i]!=null && nameFileToPlot[i] !="") {

                readFile(nameFileToPlot[i]);
            }
        }
        // pass date to Fragment
        viewModel.setMessages(messageList);

        singleRecord.clear();
        //start Fragment
        getSupportFragmentManager().beginTransaction()
                .setReorderingAllowed(true)
                .replace(R.id.flFragment, PlotFragment.class,  args: null)
                .commit();

```

Shared Model

Formed from to simple methods to set and return the *ArrayList* of the values with which we want to build the graph.

```
public class SharedModel extends ViewModel {
    private ArrayList<MessageFormat> messageList =new ArrayList<~>();


    public void setMessages(ArrayList<MessageFormat> list) { this.messageList= list; }

    public ArrayList<MessageFormat> getMessage() { return messageList ; }


}
```

Plot Fragment

This fragment plots the histography for the resistance values of the different records.

It used the libraries *MPAndroidChart* to dynamically build the plot. Initially the different values are passed through the *ShareModel*.

```
  shareModel = new ViewModelProvider(requireActivity()).get(SharedModel.class);
 messageList=  shareModel.getMessage();
```

With the contents of SharedModel, the graph Is initialized

```
ArrayList<BarEntry> resistance  = new ArrayList<>();
for( int i = 0 ; i < messageList.size(); i ++) {

    if ( messageList.get(i).getR() == Double.POSITIVE_INFINITY)
        resistance.add(new BarEntry(i,rInfinity ));
    else
    resistance.add(new BarEntry(i,(float)messageList.get(i).getR()));
}

BarDataSet barDataSet = new BarDataSet(resistance, label: "Resistance");
barDataSet.setColors(ColorTemplate.MATERIAL_COLORS);
barDataSet.setValueTextColor(Color.BLACK);
barDataSet.setValueTextSize(16f);


barData = new BarData(barDataSet);
```

In *OnCreateView* the layout and graphic are set an made available.

```java
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    View v = inflater.inflate(R.layout.fragment_plot, container,  attachToRoot: false);
    barChart = v.findViewById(R.id.barChart);
    barChart.setFitBars(true);
    barChart.setData(barData);
    barChart.getDescription().setText("Bar Chart");
    barChart.animateY( durationMillis: 2000);
    return v;


}
```