PokeType

This project was created by ©Andreas Giannakakis

Inception of the Project

My main concern before I tackled this specific idea, was to learn more about Java and Android studio. A couple of months after my graduation, I needed something new to occupy myself with, while unemployed and in search of job. With that in mind I sought to start my first Android Studio project.

The main aspects of development I knew I had to learn and master were: databases, data manipulation for the end user and the creation of an elegant UI alongside the coding problems that would occur.

I was attracted to the idea of developing an App where users could list pictures that they would like keep contained. After being asked by friends to develop an application where they could form various lists for their game, in order to remember their trade sessions, I agreed and started looking into the means of implementation thinking that, this application could provide the backbone for a more sophisticated project in the future.

Case

The problem with trades is that each player needs different monsters for their collection, so remembering each individual's selection list is close to impossible. The stimulus for this App came upon seeing the "primitive" methods of keeping their lists and passing it to each other. From .txt files to pictures taken from pages written in pencil.

Development and Implementation

The waterfall methodology was used for the development stages of the project. While the process was not strictly following the methodology, the cycle included the basic phases: Plan, Implement and Run, Test the Implementation, Resolve the bugs, errors or logical failures, Improve.

Breakdown Structure

The project can be categorized into two main sessions:

a) **Back-End**: Databases, Internal sources for data search and insertion, sorting algorithms, communication between activities and fragments.
b) **Front-End**: UI creation via Android Studio Layout Editor, Android Button Maker (Integrated XML http://angrytools.com/android/button/), Photoshop.

Databases: SQLite was used for the creation of local databases.

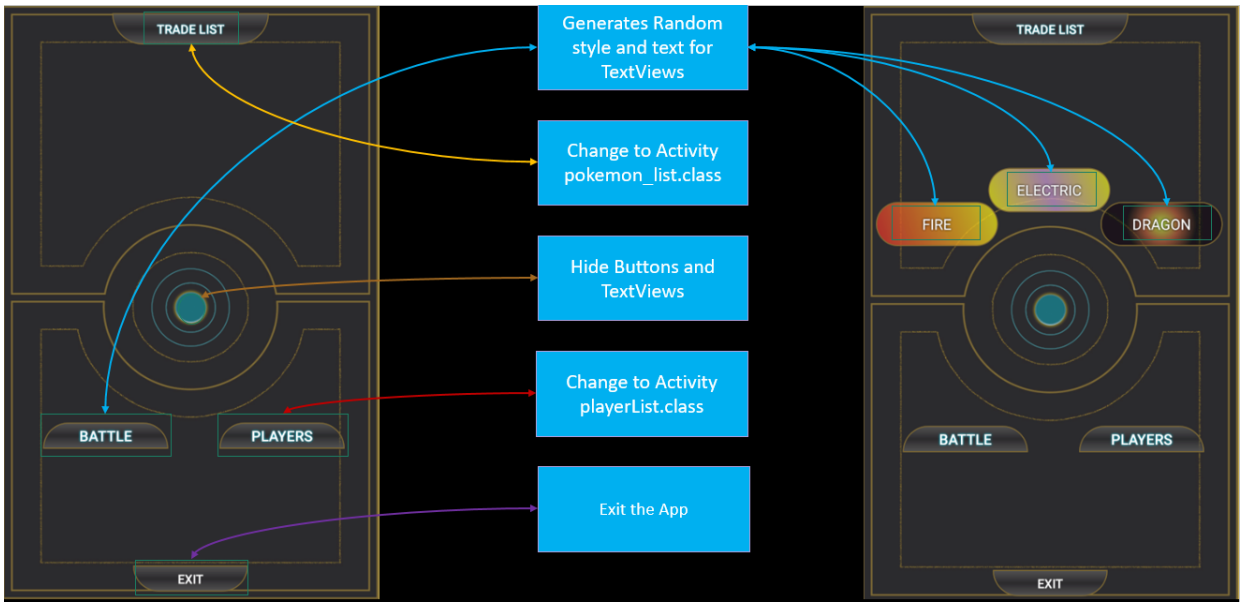
Internal Sources: .txt files, images embedded into the project.

Sorting Algorithms: Algorithms used to search for the integrated files, mentioned above, and store them in arrays, display them on Spinners, EditTexts, Grids and as backgrounds.

Communication Between Activities: Essential process for data transition to front end. For activity to fragments communication.

Activities and Java Classes

1) MainActivity.java: The home activity the user will see upon opening the app. A simple activity having five buttons.



The 'BATTLE' button follows a simple random number generator implementation.

```
Random num = new Random();
```

As for the `TextView` displays, the multitude of random numbers ranges from 0-17 for a total of 18.

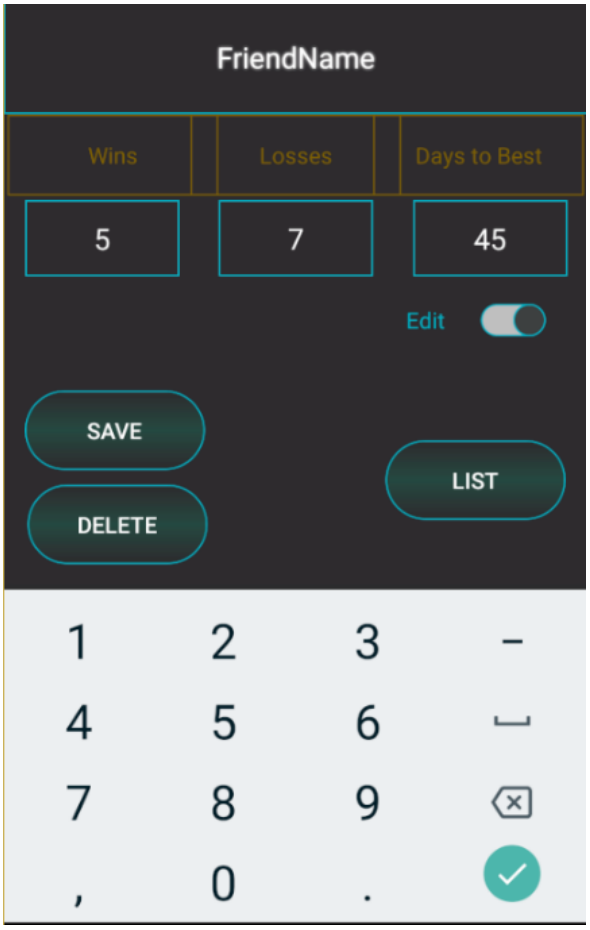
```
firstType = num.nextInt(18);
secondType = num.nextInt(18);
thirdType = num.nextInt(18);
```

Each one of the three variables shown above are processed by the switch-case function individually.

If at least two of them equals, the process is canceled and the algorithm auto-repeats until all random numbers are unequal before applying the styles and texts to the view.

```
if(firstType == secondType || secondType == thirdType || thirdType ==
firstType){
    firstType = num.nextInt(18);
    secondType = num.nextInt(18);
    thirdType = num.nextInt(18);
}


// Prevent TextViews to hold the same random number
if(firstType != secondType && secondType != thirdType && thirdType !=
firstType)
```

2) playerList.java: In this activity the user can insert, store, update and delete player names alongside their specified values.



On pressing the ‘SAVE’ button the user inserts the name typed in the `EditText` field as a row in the PlayerList.db (derives from DatabaseHandler.java), and as tables in the PlayerImage.db, PlayerShiny.db, PlayerLucky.db (each one has it’s respective classes in the project).

```
//insert data and create table if text(x) != text
if (!compareEditTextString.equalsIgnoreCase(compareCursorString) &&
!compareEditTextString.isEmpty() &&
        !compareEditTextString.equalsIgnoreCase(compareDuplicateInDB) &&
!checkForSymbol)
{
    InsertData();
    createTable();
}
```

Insert data to database:

```
boolean isInserted = myDB.insertData(
        editname.getText().toString().trim(),
        editwin.getText().toString().trim(),
        editloss.getText().toString().trim(),
        editdays.getText().toString().trim(),
        editteam.getText().toString().trim()
);

if (isInserted) { Toast.makeText(playerList.this, "Data Inserted",
Toast.LENGTH_SHORT).show();}
else { Toast.makeText(playerList.this, "Data Congestion",
Toast.LENGTH_SHORT).show();}

PopulateSpinner();
```

Create table based on name given in `EditText` field:

```
public void createTable()
{
    plrImageDB.addTable(editname.getText().toString().trim());
    plrShinyDB.addTableS(editname.getText().toString().trim());
    plrLuckyDB.addTableL(editname.getText().toString().trim());
}
```

In case the name in the EditText field already exists the ‘SAVE’ button updates the data:

```
//update data if text(x) == text(x) && not null
if (compareEditTextString.equalsIgnoreCase(compareCursorString) &&
!compareEditTextString.isEmpty() && !checkForSymbol)
{
    UpdateData();
}
```

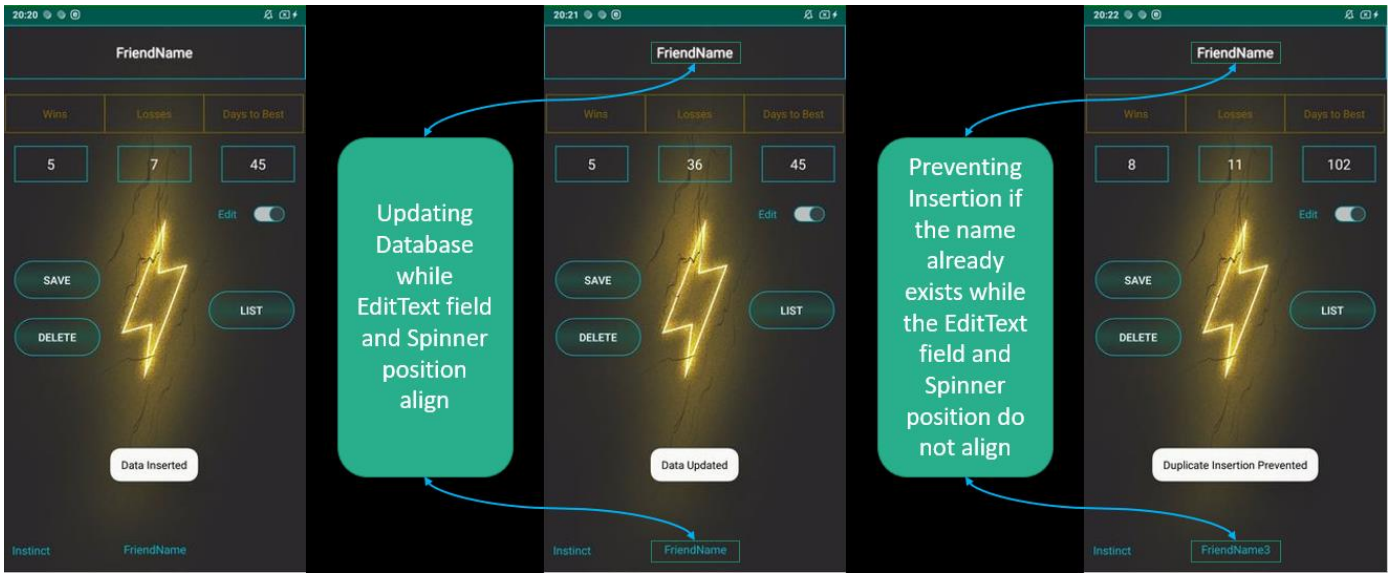
How the button works:

There are two things to keep in mind. First the data is loaded based on the string in the `EditText` field, and secondly the string in the spinner must align with the string in the `EditText` for update and deletion. All names that are saved in the database are loaded in a string array for comparison.

Case of new insertion: If the spinner is empty, then the we have a simple insertion. When the spinner gets populated by loading the data from the database, comes the need for checks and string comparisons.

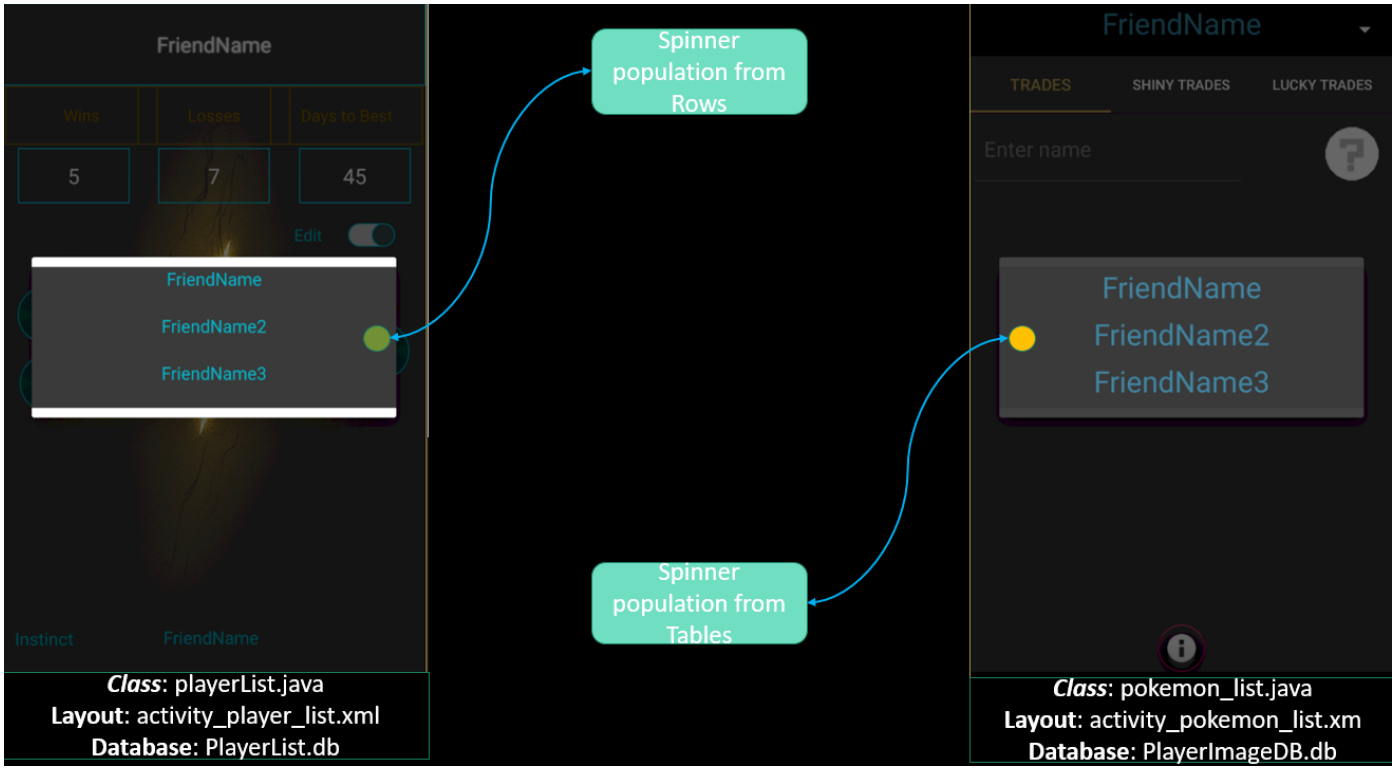
Case of pre-existing string insertion: As aforementioned the `EditText` field and spinner position must hold the same string value. In case the strings differ then we have a new insertion. If the strings are the same then an update. If the `EditText` field is empty, then the insertion of empty string is prevented and we have a pop-up message declaration. Lastly, if the `EditText` field and the string taken from the spinner position are not equal, but the string already exists in the spinner (meaning the database as well) the insertion is prevented. In order to prevent a duplicate insertion, the names of the database are also stored in a string array for the comparison of all strings that are listed.

```
//For comparison editname string with string in cursor as fetched in
LoadData function
String compareEditTextString = editname.getText().toString();
String compareCursorString = searchRow;


DatabaseHandler db = new DatabaseHandler(getApplicationContext());
List<String> compare = new ArrayList<String>();
compare = db.getColumnNames();


//transfers List<String> values to String Array Object[]
Object[] stringArray = compare.toArray();
String compareDuplicateInDB = null;


//compares string from EditText to Object[] -> (string array)
for (int i = 0; i < stringArray.length; i++)
{
    String ignoreCase = stringArray[i].toString();
    if (compareEditTextString.equalsIgnoreCase(ignoreCase))
    {
        compareDuplicateInDB = stringArray[i].toString();
    }
}
```

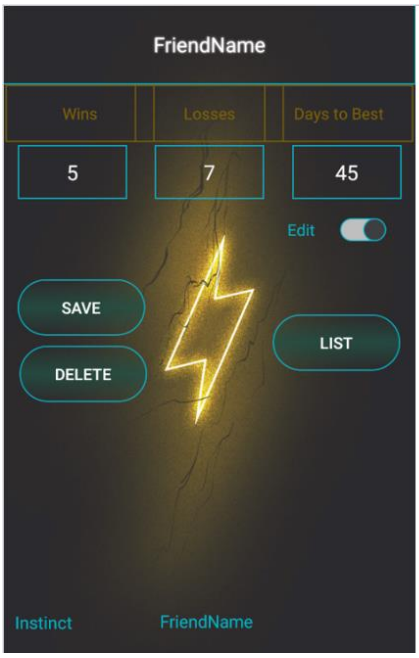


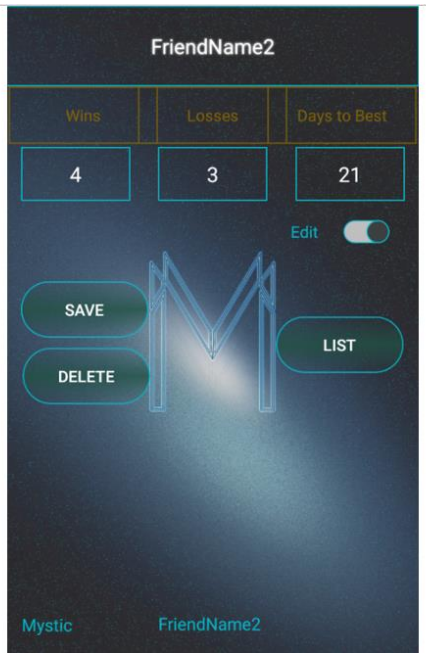
The rest of the `EditText` fields and the team name (corresponds to the background that will appear after name selection), located in another spinner, bottom left, are stored in the PlayerList.db as well.

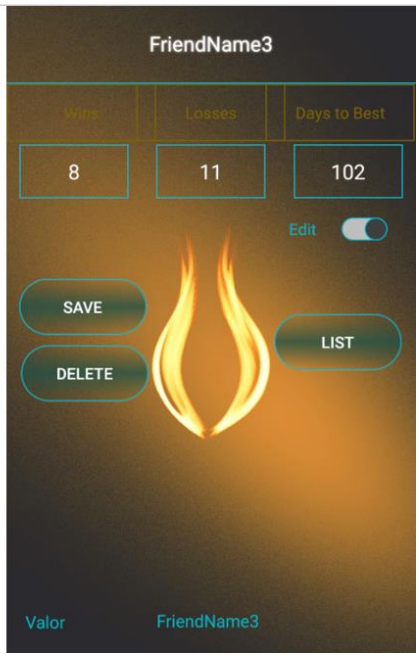
The names from both databases are loaded into spinners for selection, data update and data load.



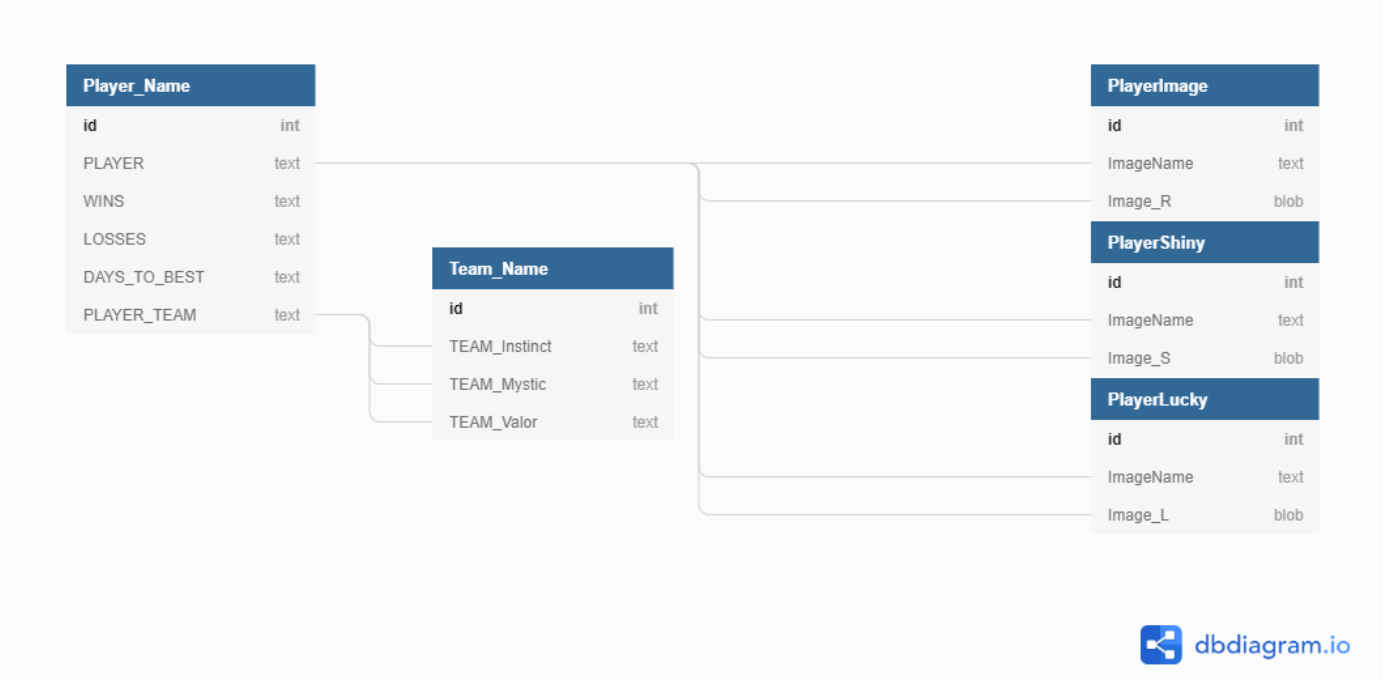
Example of 3 different entries and how they appear after selection from spinner.







Database Diagram:



There are four main databases that are being used for the app:

a) PlayerList.db where Tables = Player_Name + Team_Name.

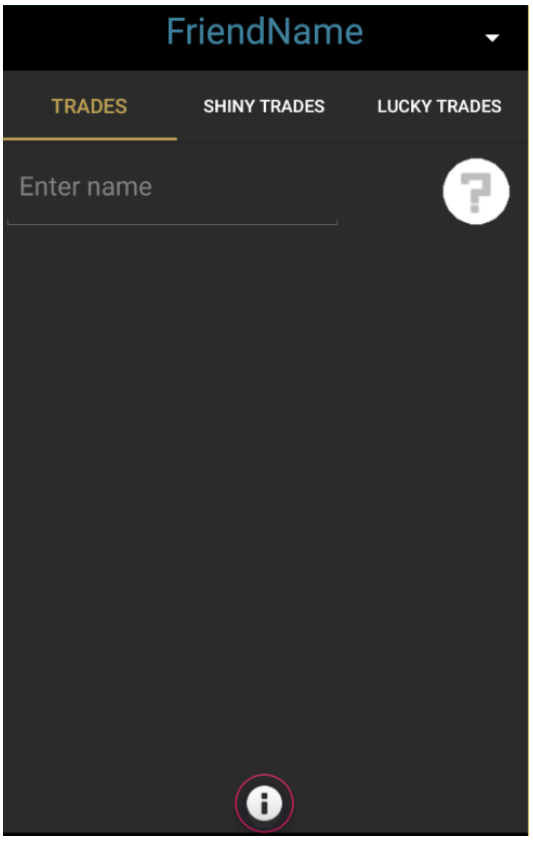
b) PlayerImage.db where Tables = “PLAYER” column from PlayerList.db.

c) PlayerShiny.db where Tables = “PLAYER” column from PlayerList.db.

d) PlayerLucky.db where Tables = “PLAYER” column from PlayerList.db.

In the PlayerList.db the names of the players are saved in rows, in parallel the same names are inserted as tables in the PlayerImage.db, PlayerShiny.db and PlayerList.db. The latter databases are used to store images for individual players on fragments.

3) pokemon_list.java: The base activity where fragments are set.



The activity is consisted by a fragment container, a spinner, the AppBarLayout, a floating button and the default `ViewPager` widget.

The Fragments: The three fragments are identical to their source code with a difference in some variable names. For the explanation the **RegularTrades.java** fragment class will be discussed.

The fragment contains the `GridView` in which the images are displayed. Each image set is created by the user and is saved accordingly to the player name that is selected from the spinner. As said above the spinner is attached to the base activity (pokemon_list.java) which parents the fragments, meaning that there is no direct communication of the spinner to the fragment code.

The traditional method of transferring data between activities and fragments is through bundles. This is not the case here. The data communication between the activity and the fragment is processed through a third class **SectionsPagerAdapter.java.**

Code from pokemon_list.java class

```
SectionsPagerAdapter sectionsPagerAdapter = new SectionsPagerAdapter(this,
getSupportFragmentManager(), value);


@Override
public void onItemSelected(AdapterView<?> parent, View view, int position,
long id)
{
    //Sets spinner selected name to string from PlayerForGrids dropdown
menu and passes values
    value = parent.getItemAtPosition(position).toString();//get spinner
text position
    ((SectionsPagerAdapter)viewPager.getAdapter()).setValue(value);
}
```

The variable `value` stores the string contained in the x position of the spinner and passes it to the SectionsPagerAdapter class.

Then in the SectionsPagerAdapter.java `value` is initialized:

```
private String Value;
```

```
public SectionsPagerAdapter(Context context, FragmentManager fm, String
value)
{
    super(fm);
    mContext = context;
    Value = value;
}
```

and stored:

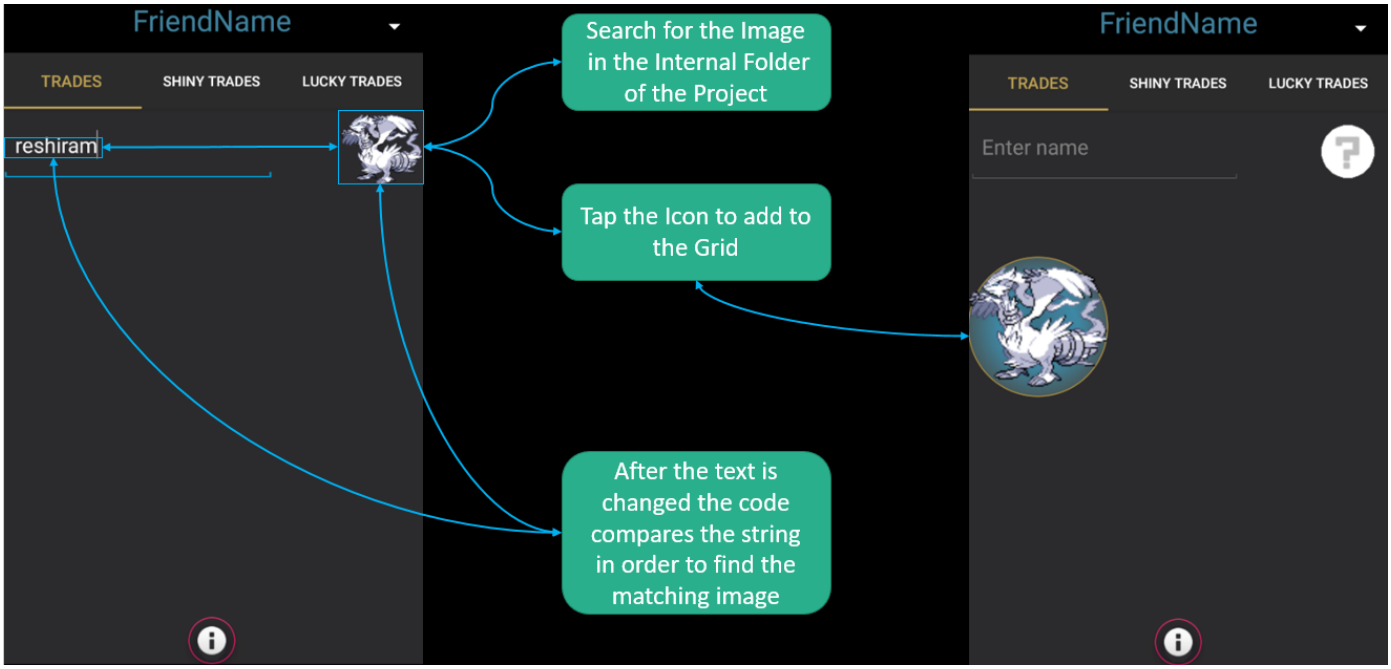
```
public void setValue(String v)
{
    this.Value = v;
    notifyDataSetChanged();
}
```

4) RegularTrades.java: This is the class of the fragment that will be analyzed. Before we reach to the point where the user can save and load different images to their player list there is the need of a constructor for the fragment.

```
//Gets Value from SectionsPagerAdapter and places into this constructor
public RegularTrades(String text)
{
    this.name = text;
}
```

As described in the comment the `Value` string from the SectionPagerAdapter is passed to `String text` and then to `String name` which is initialized inside the fragment activity. This is an important step for the cycle of the app as each time the user changes the position of the spinner, the new text will be fetched in this constructor. Then the *String* `nameText` (`nameText.setText(name);`) is used for table identification from the databases.

Now that the program can distinguish the table names the user can start inserting data in the tables.



Understanding the procedure.

Linear alignment of names and images: The names are loaded in a database from a .txt file (**pokemon_list.java - public void String_ImageToDatabase()**). The image files follow a numerical order in their file name (001name.png, 002name.png, etc.). This way the positions of the names in the .txt file/database and the image files in their respective folders align, and can be easily accessed with the use of arrays.

Text Comparison after character type: While the user types in the `EditText` field the program runs an algorithm for string comparison after each *char* insertion. When a String that exists in the database is recognized, the corresponding image automatically appears in an ImageView (upper right corner). The String and the Image are stored into two variables, *String* `DatabaseSaveNames` and *byte[x]* `DatabaseSaveImages`, where *x* = the position that derives from the index of the array that the filtered String is

found. The aforementioned variables store the values and prepare them for insertion to their selected table.

```
@Override
public void afterTextChanged(Editable s)
{
    //used to set images in ImageView according to text name
    String getName = pkmnName.getText().toString();

    //used to pass names from PlayerImageDB.db Table to Grid
    PlayerImageDB dbs = new PlayerImageDB(getActivity());

    //Get the names that respond to each image from the player tables
    List<String> pkmnNames = new ArrayList<String>();
    pkmnNames = dbs.getNamesForGrid(nameText.getText().toString());

    //Pass the names to array for comparison and automated image selection
through cursor
    Object[] imageNames = pkmnNames.toArray();

    for(int i = 0; i < stringNames.length; i++)
    {

        if(getName.equalsIgnoreCase(stringNames[i].toString()))
        {

            pokemonimage.setBackground(drawables[i]);

            //convert Drawble to byte[]
            Bitmap bitmap = ((BitmapDrawable)drawables[i]).getBitmap();
            ByteArrayOutputStream stream = new ByteArrayOutputStream();
            bitmap.compress(Bitmap.CompressFormat.PNG, 90, stream);

            //prepares values for insertion to Database
            DatabaseSaveNames = stringNames[i].toString();
            DatabaseSaveImages = stream.toByteArray();
            spritesButton.setEnabled(true);
        }
}

public void SaveImagestoGrid()
{
    //save selection on button click
    plrImageDB.insertStringImageToTable(name, DatabaseSaveNames,
DatabaseSaveImages);
}
```

Grid Insertions: In the grid the saved images of the tables which represent the Player Names are loaded. In order to store and add to grid three classes are needed:

a) **Item.java**: The creation of a simple class constructor for the Images

```
public class Item
{

    Drawable image;

    public Item(Drawable Pimage)
    {
        this.image = Pimage;
    }

    public Drawable getPokImage()
    {
        return image;
    }

}
```

b) **RegularpkmnAdapter.java**: The `ArrayAdapter<Item>` adapter attached to the `GridView`. Responsible for containing the elements of the grid in an `ArrayList<Item>`. The `<Item>` originates from the **Item.java**.

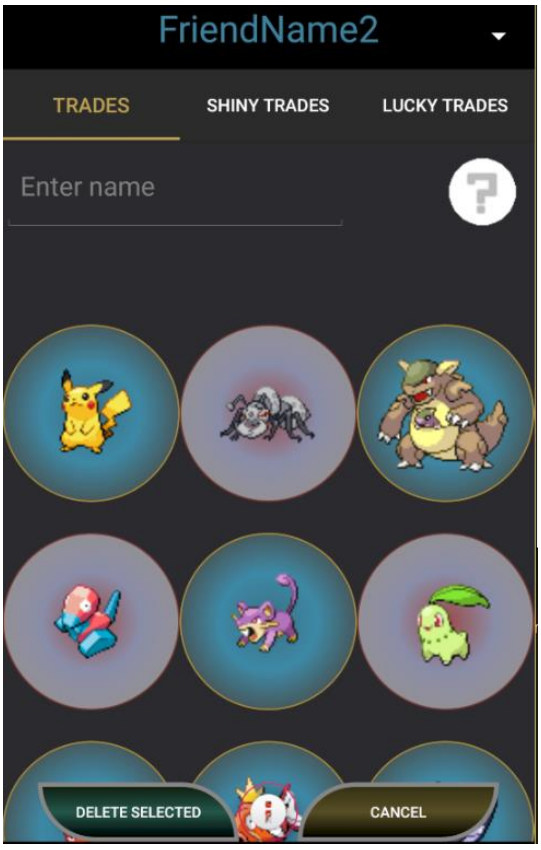
```
public class RegularpkmnAdapter extends ArrayAdapter<Item>
{

    ArrayList<Item> gridimages = new ArrayList();
    public List selectedPositions;


    public RegularpkmnAdapter(Context context, int itemView,
ArrayList<Item> objects)
    {
        super(context, itemView, objects);
        gridimages = objects;
        selectedPositions = new ArrayList<>();
}
```

For the Image and the background view inside the grid's elements, `R_Grid_View` from **R_Grid_View.java**, is used.

```
@Override
public View getView(int position, View convertView, ViewGroup parent)
{
        R_Grid_View customView = (convertView == null) ? new
R_Grid_View(getContext()) : (R_Grid_View) convertView;
        customView.display(gridimages.get(position).getPokImage(),
selectedPositions.contains(position));
    return customView;
}
```

c) **R_Grid_View.java**: Displays background in the grid's elements. Used for changing the background upon position selection in order to notify the user for their selected items.

The gray-red background indicates that the item is selected



Back in the **RegularTrades.java** the objects mentioned in the previous classes need to be linked to the main functions of the code. After the initialization of the grid and grid adapter the images saved in the database are loaded.

```
//pass names from PokemonDatabase.db Table to Grid
    plrImageDB = new PlayerImageDB(getActivity());

    //Get the names that respond to each image from the database
    List<String> pkmnNames = null;
    pkmnNames = plrImageDB.getNamesForGrid(nameText.getText().toString());

    //Pass the names to array for comparison and automated image selection
through cursor
    Object[] imageNames = pkmnNames.toArray();
    gridnames = pkmnNames.toArray(new String[0]);
    gridimages = new ArrayList<>();

    byte[] imagesGr = null;

    for(int i = 0; i < imageNames.length; i++)
    {
        String name = imageNames[i].toString();
        Cursor cursor =
plrImageDB.getImageforGrid(nameText.getText().toString(), name);

        if (cursor.moveToFirst())
        {
            imagesGr = cursor.getBlob(0);
            Drawable imageR = new BitmapDrawable(getResources(),
BitmapFactory.decodeByteArray(imagesGr, 0, imagesGr.length));
            gridimages.add(new Item(imageR));
        }

    }// ends for
}//ends if

gridAdapter = new RegularpkmnAdapter(getContext(),
R.layout.regularpkmn_item, gridimages);
regularPkmnGrid.setAdapter(gridAdapter);
```

Improvements, bugs and possible fixes

This section serves as a note on the code lengths that can be either optimized or completely changed.

1) The insertion of names in the databases can be processed through HashMaps and/or classes. With this method there can be a reduction to the code written, as HashMaps auto-compare Strings and prevent duplicate insertions.

2) Saving the image path instead of a blob in the databases. Although the images are sort in size (50kb maximum), the best practice is to store the path.

Last Edit: Sep/13/2020