Why should we use RxJava on Android

Reactive Extensions (Rx) are a set of interfaces and methods which provide a way to developers solve problems rapidly, simply to maintain, and easy to understand. RxJava provides just that, a set of tools to help you write clean and simpler code.

To be honest with you, at first I thought RxJava was just way to difficult to understand and the ideia to add a library just to use these new tools was very troubling for me. After awhile, I understood that I was struggling by doing the same boilerplate implementation over and over again to keep the user updated on what is going on the application.

The amount of work I've done refactoring all necessary methods and interfaces just because there was a new requirement, or even changing the way information was presented or was processed, was just ridiculous. Additionally, understanding all that amount of code, probably done by someone else was very time consuming.

Lets imagine that we need to fetch a list of users from the database and then present those results on our view. For this we could make an AsyncTask to fetch all information from the database and then insert it on your our adapter to present the results to the user. For the purposes of this example lets make it a simple task:

```
public class SampleTask extends AsyncTask<Void,Void,List<Users>> {
    private final SampleAdapter mAdapter;

    public SampleTask(SampleAdapter sampleAdapter) {
        mAdapter = sampleAdapater;
    }

    @Override
    protected List<Users> doInBackground(Void... voids) {
        //fetch there results from the database and return them to the
onPostExecute
        List<Users> users = getUsersFromDatabase();
        return users;
    }

    @Override
    protected void onPostExecute(List<Users> users) {
        super.onPostExecute(products);
        // Checking if there are users on the database
        if(users == null) {
```

```
            //No users, presenting a view saying there are no users
            showEmptyUsersMessageView();
            return;
        }

        mAdapter.addAll(users);
        mAdapter.notifyDataSetChanged();
    }
}
```

If there is a new requirement saying that should only appear the non Guest users, the approach would be to add a condition before adding it to the adapter, or changing the database query. Furthermore, imagine if instead of fetching only information from users you now have a requirement that you need to fetch something else from the database to present on the same adapter?

That is why we have RxJava which can help us a lot in these situations. For example, we can have something like this, in order to better structure our code:

```
public Observable<List<User>> fetchUsersFromDatabase() {
    return Observable.create(new Observable.OnSubscribe<List<User>>(){
        @Override
        public void call(Subscriber<? super List<User>> subscriber){
            // Fetch information from database
            subscriber.onNext(getUserList());
            subscriber.onCompleted();
        }
    });
}
```

Then the it can be called like this:

```
fetchUsersFromDatabase()
    //will process everything in a new thread
    .subscribeOn(Schedulers.io())
    //will listen the results on the main thread
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Subscriber<List<User>>() {
        @Override
        public void onCompleted() {

        }

        @Override
        public void onError(Throwable e) {
```

```
        }

        @Override
        public void onNext(List<User> users) {
            //Do whatever you want with each user
        }
    });
}
```

You must be thinking, and if we want to get all the users except Guests? Well with this approach its quite easy to change it, the developer only needs to add a filter to it.

```
fetchUsersFromDatabase()
        .filter(new Func1<User, Boolean>() {
            @Override
            public Boolean call(User user) {
                //only return the users which are not guests
                return !user.isGuest();
            }
        })
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Subscriber<User>() {
                       @Override
                       public void onCompleted() {
                       }

                       @Override
                       public void onError(Throwable e) {
                           /*Check if there was any error while retriev
ing from database*/
                       }

                       @Override
                       public void onNext(User user) {
                           //Do whatever you want with each user
                       }
                   }
        );
```

A simple operation of transforming or filtering the information from the database would require new interfaces and restructuring code in order to respect the implemented architecture. With RxJava this is a lot easier, by simply creating an Observable which retrieves all the information and then you can use any of these methods to filter and retrieve only the information you want.

Although you must be saying Ok, it's nicer to read and to structure, but this approach creates so much code. Well you are right, but here is where Retrolambda shines. This library which provides compatibility to use Java 8 lambda expressions, method references, and so much more.

Using this library will help reduce the code significantly:

```
fetchUsersFromDatabase()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(value -> {
                    //Do whatever with the value
                },error -> {
                    //do something with in case of error
                }
        );
```

And you must be asking what should we do if we have to run other query to present it on the same adapter? Well we can do something like this:

```
fetchUsersFromDatabase()
        .zipWith(fetchSomethingElseFromDatabase(), (users, somethingEls
e) -> {/*here combine users and something else into a new object*/})
        .subscribe( o -> {/*use the combine object from users and somet
hing else to fill the adapter */});
```

In this example, we join the results from users and something else from the database and then we can fill the adapter. Much easier to maintain, less code and easy to read right?

In order to have a deeper look on the many available methods and ways the information can be retrieved have a look at this article it should help you. Additionally it might help you setting everything up.