Chapter 1

RxJava Introduction

What is RxJava?

Why RxJava?

Let's start in fashion - Life will be easy with RxJava.

RxJava is an art and endless possibilities await those who can master it.

Yes, that's true.

So, let's see technically what is RxJava?

RxJava is a Java VM implementation of Reactive Extensions.

Question arises in mind, What is Reactive Extensions?

From the official doc, Reactive Extensions(ReactiveX) is as a library for composing **asynchronous** and **event-based** programs by using **observable sequences**.

Now we have to learn these three terms - **asynchronous**, **event-based** and **observable sequences**.

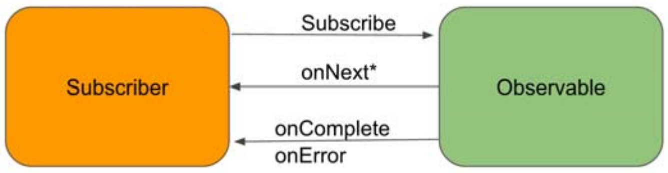
1. **Asynchronous:** Different parts of the program run simultaneously.
2. **Event-Based:** The program executes the code based on the event generated.
3. **Observable Sequences:** We will see it later for better understanding.

4. **The Reactive Extension is further elaborated:** It extends the observer pattern to support sequences of data and/or events and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O.

> *Here, observer pattern is the key to understand the architecture, which is a software design pattern, where Subject maintains its Observers and notifies them automatically of any state change, usually by calling one of their methods.*

RxJava is an implementation of Reactive Stream Specification

How does this Reactive Stream implement?



There are two main components of RxJava:

1. **Observable**: it emits the values.
2. **Observer**: it gets the values.

Observer get the emitted values from Observable by subscribing on it.

All components are as follows:

1. **Flowable, Observable, Single, and Completable** - does some work and emit values.
2. **Subsciption** - work is going on or completed or is used to cancel.
3. **Operators** - Modify Data

4. **Schedulers** - Where the work should be done, which thread like main thread, etc.
5. **Subscriber/Disposable** - where the response will be sent after work has been completed.

Show me the code to understand it in a better way.

```java
// Observable just is a factory method that emmits the values.
private Observable<String> getObservable() {
    return Observable.just("Cricket", "Football");
}
```

```java
private Observer<String> getObserver() {
    return new Observer<String>() {

        @Override
        public void onSubscribe(Disposable d) {

        }

        @Override
        public void onNext(String value) {
            System.out.println(value);
        }

        @Override
        public void onError(Throwable e) {

        }

        @Override
        public void onComplete() {
```

```
                    System.out.println("onComplete");
                }
            };
        }
```

Now, we have to connect both **Observable** and **Observer** with a **Subscription**. Only then can it actually do anything:

```
        private void doSomeWork() {
            getObservable()
                    // Run on a background thread
                    .subscribeOn(Schedulers.io())
                    // Be notified on the main thread
                    .observeOn(AndroidSchedulers.mainThread())
                    .subscribe(getObserver());
        }
```

This will print the following

Cricket

Football

onComplete

This means the **Observable** emits two strings, one by one, which are observed by **Observer**.

So, here it goes to onNext two times and then after emitting all the values it goes to onComplete

The complete RxJava Examples can be found here: Link

Types of Observables

Observables produce stream to be observed by an Observer when it subscribes. There are various types and ways this stream can be emitted and it depends on the use case. Some of those use cases are as listed below:

1. Items are to be emitted one by one.
2. Items are to be emitted by controlling the producers emission(with backpressure: we will learn backpressure later in this course).
3. Only one item need to be emmited as part of success.
4. Item has to be emitted based on conditions.

There are 5 types of observables as follows:

- **Observable:** It emits 0..N elements, and then completes successfully or with an error.

The **Observer** for the **Simple** type of **Observable** will be:

```
private Observer<String> getObserver() {
    return new Observer<String>() {

        @Override
        public void onSubscribe(Disposable d) {

        }

        @Override
        public void onNext(String value) {

        }

        @Override
```

```
            public void onError(Throwable e) {

            }

            @Override
            public void onComplete() {

            }
        };
    }
```

- **Flowable:** Similar to **Observable** but with a backpressure strategy.

So, the **Observer** for the **Flowable** type of **Observable** will be:

```
    private Observer<String> getObserver() {
        return new Observer<String>() {

            @Override
            public void onSubscribe(Disposable d) {

            }

            @Override
            public void onNext(String value) {

            }

            @Override
            public void onError(Throwable e) {

            }

            @Override
            public void onComplete() {

            }
        };
    }
```

- **Single:** It completes with a value successfully or an error.(*doesn't have onComplete callback, instead onSuccess(val)*).

The **Observer** for the **Single** type of **Observable** will be:

```
    private SingleObserver<Integer> getSingleObserver() {

        return new SingleObserver<Integer>() {
            @Override
            public void onSubscribe(Disposable d) {

            }

            @Override
            public void onSuccess(Integer value) {

            }

            @Override
            public void onError(Throwable e) {

            }
        };
    }
```

- **Maybe:** It completes with/without a value or completes with an error.

The **Observer** for the **Maybe** type of **Observable** will be:

```
private MaybeObserver<Integer> getMaybeObserver() {
    return new MaybeObserver<Integer>() {
        @Override
        public void onSubscribe(Disposable d) {

        }

        @Override
        public void onSuccess(Integer value) {

        }

        @Override
        public void onError(Throwable e) {

        }

        @Override
        public void onComplete() {

        }
    };
}
```

- **Completable:** It just signals if it has completed successfully or with an error.

The **Observer** for the **Completable Observable** will be:

```
private CompletableObserver getCompletableObserver() {
    return new CompletableObserver() {
        @Override
        public void onSubscribe(Disposable d) {

        }
```

```
            @Override
            public void onComplete() {

            }

            @Override
            public void onError(Throwable e) {

            }
        };
    }
```

So, we will have to choose which one to use on basis of our use-case.

The complete RxJava Examples can be found here: Link

Schedulers

RxJava Schedulers is all about managing thread at low level. They provide high-level constructs to manage with concurrency and deal with details by itself. They create workers who are responsible for scheduling and running code. By default RxJava will not introduce concurrency and will run the operations on the subscription thread.

There are two methods through which we can introduce Schedulers into our chain of operations:

- **subscribeOn**: It specify which Scheduler invokes the code contained in Observable.create().
- **observeOn:** It allows control to which Scheduler executes the code in the downstream operators.

RxJava provides some general use Schedulers:

- **Schedulers.computation()** : Used for CPU intensive tasks.
- **Schedulers.io()**: Used for IO bound tasks.
- **Schedulers.from(Executor)**: Use with custom ExecutorService.
- **Schedulers.newThread():** It always creates a new thread when a worker is needed. Since it’s not thread pooled and always creates a new thread instead of reusing one, this scheduler is not very useful.

Home > Course > Chapter 2

Chapter 2

Operators

What are operators?

Operators are basically a set of functions that can operate on any observable and defines the observable, how and when it should emit the data stream.

Few useful operators are as follows:

1. **Map**: It transforms the items emitted by an Observable by applying a function to each item.
2. **Zip**: It combines the emissions of multiple Observables together via a specified function, then emits a single item for each combination based on the results of this function.
3. **Filter**: It emits only those items from an Observable that pass a predicate test.
4. **FlatMap**: It transforms the items emitted by an Observable into Observables, then flattens the emissions from those into a single Observable.
5. **Take**: It emits only the first n items emitted by an Observable.
6. **Reduce**: It applies a function to each item emitted by an Observable, sequentially, and emits the final value.
7. **Skip**: It suppresses the first n items emitted by an Observable.
8. **Buffer**: It periodically gathers items emitted by an Observable into bundles and emits these bundles rather than emitting the items one at a time.
9. **Concat**: It emits the emissions from two or more Observables without

9. **Concat**: It emits the emissions from two or more Observables without interleaving them.
10. **Replay**: It ensures that all observers see the same sequence of emitted items, even if they subscribe after the Observable has begun emitting items.
11. **Merge**: It combines multiple Observables into one by merging their emissions.

he best way to understand operator are by examples. So next is the examples.

Map Operator

Map transforms the items emitted by an Observable by applying a function to each item.

As an example: Lets say we are getting ApiUser Object from api server then we are converting it into User Object because may be our database support User Not ApiUser Object. Here we are using Map Operator to do that.

```
private Observable<List<ApiUser>> getObservable() {
    return Observable.fromCallable(new Callable<List<ApiUser>>() {
        @Override
        public List<ApiUser> call() throws Exception {
            return Utils.getApiUserList();
        }
    });
}
```

```
private Observer<List<User>> getObserver() {
    return new Observer<List<User>>() {

        @Override
        public void onSubscribe(Disposable d) {

        }

        @Override
        public void onNext(List<User> userList) {
            // do anything with the list of user.
        }
```

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

            }

            @Override
            public void onComplete() {

            }
        };
    }
```

```


    private void doSomeWork() {
        getObservable()
                // Run on a background thread
                .subscribeOn(Schedulers.io())
                // Be notified on the main thread
                .observeOn(AndroidSchedulers.mainThread())
                .map(new Function<List<ApiUser>, List<User>>() {

                    @Override
                    public List<User> apply(List<ApiUser> apiUsers) throw
                        return Utils.convertApiUserListToUserList(apiUser
                    }
                })
                .subscribe(getObserver());
    }
```

So, this is how Map operator can be used to map an object into an another object.

Zip Operator

Zip combines the emissions of multiple Observables together via a specified function, then emits a single item for each combination based on the results of this function

As an example: Lets say we have to make two network call one for getting the list of users who loves cricket, another for users who loves football. And then returns the list of user who loves both. Here we are using Zip Operator to do that.

```
/*
 * This observable return the list of User who loves cricket
 */
private Observable<List<User>> getCricketFansObservable() {
    return Rx2AndroidNetworking.get("url")
            .build()
            .getObjectListObservable(User.class);
}

/*
* This observable return the list of User who loves Football
*/
private Observable<List<User>> getFootballFansObservable() {
    return Rx2AndroidNetworking.get("url")
            .build()
            .getObjectListObservable(User.class);
}

/*
*  Logic to filter user who loves both
*/
private List<User> filterUserWhoLovesBoth(List<User> cricketFans, Lis
    List<User> userWhoLovesBoth = new ArrayList<>();
    for (User cricketFan : cricketFans) {
            for (User footballFan : footballFans) {
                if (cricketFan.id == footballFan.id) {
                    userWhoLovesBoth.add(cricketFan);
                }
            }
        }
        return userWhoLovesBoth;
    }

    /*
    * This do the complete magic, make both network call
    * and then returns the list of user who loves both
    * Using zip operator to get both response at a time
    */
    private void findUsersWhoLovesBoth() {
        // here we are using zip operator to combine both request
        Observable.zip(getCricketFansObservable(), getFootballFansObserva
                new BiFunction<List<User>, List<User>, List<User>>() {
                    @Override
                    public List<User> apply(List<User> cricketFans, List<
                            throws Exception {
                        List<User> userWhoLovesBoth =
                                filterUserWhoLovesBoth(cricketFans, footb
                        return userWhoLovesBoth;
                    }
                })
                .subscribeOn(Schedulers.newThread())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Observer<List<User>>() {
                    @Override
                    public void onSubscribe(Disposable d) {

                    }

                    @Override
                    public void onNext(List<User> users) {
                        // do anything with user who loves both
                    }


                    @Override
                    public void onError(Throwable e) {

                    }

                    @Override
                    public void onComplete() {

                    }
                });
    }
```

So, this is how Zip operator can be used to combine two or more tasks.

The complete RxJava Examples can be found here: Link

FlatMap with Filter Operator

FlatMap transforms the items emitted by an Observable into Observables, then flattens the emissions from those into a single Observable.

Filter emits only those items from an Observable that pass a predicate test.

As an example: Let say our server return the list of my friends , but we need to filter out only who those friends who is also following me. Here comes the filter operator to do so.

```
private Observable<List<User>> getAllMyFriendsObservable() {
    return Rx2AndroidNetworking.get("url")
            .build()
            .getObjectListObservable(User.class);
}

public void flatMapAndFilter() {
    getAllMyFriendsObservable()
            .flatMap(new Function<List<User>, ObservableSource<User>>
                // flatMap - to return users one by one
                @Override
                public ObservableSource<User> apply(List<User> usersL
                    return Observable.fromIterable(usersList);
                    // returning user one by one from usersList.
                }
            })
            .filter(new Predicate<User>() {
                @Override
                public boolean test(User user) throws Exception {
                    // filtering user who follows me.
                    return user.isFollowing;
                }

            })
```

```
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Observer<User>() {
                    @Override
                    public void onSubscribe(Disposable d) {

                    }

                    @Override
                    public void onNext(User user) {
                        // only the user who is following me comes here o
                    }

                    @Override
                    public void onError(Throwable e) {

                    }

                    @Override
                    public void onComplete() {

                    }
                });
    }
```

This way we can use filter and flatMap operators.

The complete RxJava Examples can be found here: Link

FlatMap with Zip Operator

FlatMap transforms the items emitted by an Observable into Observables, then flattens the emissions from those into a single Observable.

Zip combines the emissions of multiple Observables together via a specified function, then emits a single item for each combination based on the results of this function.

As an example: Let say our server return the list of the users , then we have to get the user detail of each corresponding user.

```
private Observable<List<User>> getUserListObservable() {
    return Rx2AndroidNetworking.get("url")
            .build()
            .getObjectListObservable(User.class);
}

private Observable<UserDetail> getUserDetailObservable(long id) {
    return Rx2AndroidNetworking.get("url")
            .build()
            .getObjectObservable(UserDetail.class);
}

public void flatMapWithZip(View view) {
    getUserListObservable()
            .flatMap(new Function<List<User>, ObservableSource<User>>
                // flatMap - to return users one by one
                @Override
                public ObservableSource<User> apply(List<User> usersL
                    return Observable.fromIterable(usersList);
                    // returning user one by one from usersList.
                }

            })
```

```
                .flatMap(new Function<User, ObservableSource<Pair<UserDet
                    @Override
                    public ObservableSource<Pair<UserDetail, User>> apply
                        // here we get the user one by one and then we ar
                        // two observable - one getUserDetailObservable (
                        // and another Observable.just(user) - just to em
                        return Observable.zip(getUserDetailObservable(use
                                Observable.just(user),
                                new BiFunction<UserDetail, User, Pair<Use
                                    @Override
                                    public Pair<UserDetail, User> apply(U
                                            throws Exception {
                                        // runs when network call complet
                                        // we get here userDetail for the
                                        return new Pair<>(userDetail, use
                                        // returning the pair(userDetail,
                                    }
                                });
                    }
                })
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Observer<Pair<UserDetail, User>>() {
                    @Override
                    public void onComplete() {
                        // do something onCompleted
                    }

                    @Override
                    public void onError(Throwable e) {
                        // handle error
                        Utils.logError(TAG, e);
                    }

                    @Override
                    public void onSubscribe(Disposable d) {

                    }


                    @Override
                    public void onNext(Pair<UserDetail, User> pair) {
                        // here we are getting the userDetail for the
                        // corresponding user one by one
                    }
                });
    }
```

This way we can use zip and flatMap operators.

The complete RxJava Examples can be found here: Link

Chapter 3

RxJava Advance

Understanding RxJava Subject — Publish, Replay, Behavior and Async Subject

What is Subject?

> *A Subject is a sort of bridge or proxy that is available in some implementations of ReactiveX that acts both as an observer and as an Observable. Because it is an observer, it can subscribe to one or more Observables, and because it is an Observable, it can pass through the items it observes by re-emitting them, and it can also emit new items.*

We believe : **learning by examples is the best way to learn**

There are four types of Subject available in RxJava.

- Publish Subject
- Replay Subject
- Behavior Subject
- Async Subject

Observable: Assume that a professor is an observable. The professor teaches about some topic.

Observer: Assume that a student is an observer. The student observes the topic being taught by the professor.

Publish Subject

It emits all the subsequent items of the source Observable at the time of subscription.

Here, if a student entered late into the classroom, he just wants to listen from that point of time when he entered the classroom. So, `Publish` will be the best for this use-case.

See the below example:

```
PublishSubject<Integer> source = PublishSubject.create();

// It will get 1, 2, 3, 4 and onComplete
source.subscribe(getFirstObserver());

source.onNext(1);
source.onNext(2);
source.onNext(3);

// It will get 4 and onComplete for second observer also.
source.subscribe(getSecondObserver());

source.onNext(4);
source.onComplete();
```

Check the complete example here.

Replay Subject

It emits all the items of the source Observable, regardless of when the subscriber subscribes.

Here, if a student entered late into the classroom, he wants to listen from the beginning. So, here we will use `Replay` to achieve this.

See the below example:

```
ReplaySubject<Integer> source = ReplaySubject.create();
// It will get 1, 2, 3, 4
source.subscribe(getFirstObserver());
source.onNext(1);
source.onNext(2);
source.onNext(3);
source.onNext(4);
source.onComplete();
// It will also get 1, 2, 3, 4 as we have used replay Subject
source.subscribe(getSecondObserver());
```

Check the complete example here.

Behavior Subject

It emits the most recently emitted item and all the subsequent items of the source Observable when an observer subscribes to it.

Here, if a student entered late into the classroom, he wants to listen the most recent things(not from the beginning) being taught by the professor so that he gets the idea of the context. So, here we will use `Behavior`.

See the below example:

```
BehaviorSubject<Integer> source = BehaviorSubject.create();
```

```
        // It will get 1, 2, 3, 4 and onComplete
        source.subscribe(getFirstObserver());
        source.onNext(1);
        source.onNext(2);
        source.onNext(3);
        // It will get 3(last emitted)and 4(subsequent item) and onComple
        source.subscribe(getSecondObserver());
        source.onNext(4);
        source.onComplete();
```

Check the complete example here.

Async Subject

It only emits the last value of the source Observable(and only the last value)

Here, if a student entered at any point of time into the classroom, and he wants to listen only about the last thing(and only the last thing) being taught. So, here we will use `Async`.

See the below example:

```
        AsyncSubject<Integer> source = AsyncSubject.create();
        // It will get only 4 and onComplete
        source.subscribe(getFirstObserver());
        source.onNext(1);
        source.onNext(2);
        source.onNext(3);
        // It will also get only get 4 and onComplete
        source.subscribe(getSecondObserver());
        source.onNext(4);
        source.onComplete();
```

Check the complete example here.

So, whenever you are stuck with these type of cases, the RxJava Subject will be your best friend.

The complete RxJava Examples can be found here: Link

Using Disposable in RxJava

As the android applications do many operations in background, many times it can happen that the activity no longer needs the data after sometime from that operation, in that case, there must have a way to unsubscribe to avoid memory leaks.

So, the RxJava gives us the Disposable to unsubscribe to avoid memory leaks.

Lets see this with an example to understand it better. Let say there in an observable with emits some values, but while emitting values, the use presses back button, then it must stops emitting values.

```
public class DisposableExampleActivity extends AppCompatActivity {

    private final CompositeDisposable disposables = new CompositeDisposab

    @Override
    protected void onStop() {
        super.onStop();
        disposables.clear(); // do not send event after activity has been
    }

    /*
     * Example to understand how to use disposables.
     * disposables is cleared in onDestroy of this activity.
     */
    void doSomeWork() {
        disposables.add(sampleObservable()
                // Run on a background thread
                .subscribeOn(Schedulers.io())
                // Be notified on the main thread
                .observeOn(AndroidSchedulers.mainThread())
```

```
                  .subscribeWith(new DisposableObserver<String>() {
                      @Override
                      public void onComplete() {

                      }

                      @Override
                      public void onError(Throwable e) {

                      }

                      @Override
                      public void onNext(String value) {

                      }
                  }));
    }

    static Observable<String> sampleObservable() {
        return Observable.defer(new Callable<ObservableSource<? extends S
            @Override
            public ObservableSource<? extends String> call() throws Excep
                // Do some long running operation
                SystemClock.sleep(2000);
                return Observable.just("one", "two", "three", "four", "fi
            }
        });
    }
}
```

This way memory leaks can be avoided by using Disposable.

Another important thing is that there are two methods one is clear() and another is dispose().

1. **clear()** will clear all, but can accept new disposable.

2. **dispose()** will clear all and set isDisposed = true, so it will not accept any new disposable.

The complete RxJava Examples can be found here: Link

Backpressure

In cases where a publisher is emitting items more rapidly than an operator or subscriber can consume them, then the items that are overflowing from the publisher need to be handled. If for example we try to maintain an ever-expanding buffer of items emitted by the faster publisher to eventually combine with items emitted by the slower one. This could result in OutOfMemoryError.

> *Backpressure relates to a feedback mechanism through which the subscriber can signal to the producer how much data it can consume and so to produce only that amount.*

There are a number of way to handle the backpressure like buffering, batching, skipping etc. and there are operators to deal with them. These are explained in the official reference: **Link**

Let’s see some scenarios:

The Subscriber has an onSubscribe(Subscription) method, Subscription::request(long n), it is through this it can signal upstream that it’s ready to receive a number of items and after it processes the items request another batch. There is a default implementation which requests **Long.MAX_VALUE** which basically means “**send all you have**”. But we have not seen the code in the producer that takes consideration of the number of items requested by the subscriber.

Let’s see the Flowable example:

```
        Flowable.create(new FlowableOnSubscribe<String>() {
            @Override
            public void subscribe(@NonNull FlowableEmitter<String> emitte
                    throws Exception {
                int count = 0;
                while (true) {
                    count++;
                    emitter.onNext(count + "\n");
                }
            }
        },  BackpressureStrategy.DROP)
                .observeOn(Schedulers.newThread(), false, 3)
                .subscribe(new Subscriber<String>() {
                    @Override
                    public void onSubscribe(Subscription s) {

                    }

                    @Override
                    public void onNext(String value) {
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }

                    @Override
                    public void onError(Throwable t) {

                    }

                    @Override
                    public void onComplete() {

                    }
                });
```

The 2nd parameter **BackpressureStrategy** allows us to specify what to do in the case of overproduction.

- **BackpressureStrategy.BUFFER:** It buffers in memory the events that overflow. If we don't provide some threshold, it might lead to OutOfMemoryError.
- **BackpressureStrategy.DROP**: It simply drop the overflowing events
- **BackpressureStrategy.LATEST**: It keeps only recent event and discards previous unconsumed events.
- **BackpressureStrategy.ERROR**: We get an error in the Subscriber immediately.
- **BackpressureStrategy.MISSING**: We use this when we don't care about backpressure(we let one of the downstream operators onBackpressureXXX handle it)

By default the subscriber requests **Long.MAX_VALUE** since the code flowable.subscribe(onNext, onError, onComplete) uses a default onSubscribe. So unless we override onSubscribe, it would overflow. We can control this through operator **observeOn()** that make its own request to the upstream Publisher(256 by default), but can take a parameter to specify the request size.

In the above example, initially, the Subscriber requests Long.MAX_VALUE, but as the subscription travels upstream through the operators to the source Flowable, the operator observeOn subscribes to the source and requests just 3 items from the source instead. Since we used BackpressureStrategy.DROP, all the items emitted outside the 3, get discarded and thus never reach our subscriber.

There are also specialized operators to handle backpressure the onBackpressureXXX operators:

1. **onBackpressureBuffer**
2. **onBackpressureDrop**
3. **onBackpressureLatest**

```
    .onBackpressureDrop(val -> System.out.println("Dropping " + val))
```

These operators request Long.MAX_VALUE(unbounded amount) from upstream and then take it upon themselves to manage the requests from downstream. In the case of *onBackpressureBuffer* it adds in an internal queue and sends downstream the events as requested, *onBackpressureDrop* just discards events that are received from upstream more than requested from downstream, *onBackpressureLatest* also drops emitted events excluding the last emitted event(most recent). The last onBackpressureXXX operator overrides the previous one if they are chained.

The complete RxJava Examples can be found here: Link