Reactive Programming:

why Reactive programming??????

**Reason1** : 10 years ago, interaction with web pages was basically about submitting a long form to the backend and performing simple rendering to the frontend. Apps have evolved to be more real-time: modifying a single form field can automatically trigger a save to the backend, "likes" to some content can be reflected in real time to other connected users, and so forth.

Apps nowadays have an abundancy of real-time events of every kind that enable a highly interactive experience to the user. We need tools for properly dealing with that, and Reactive Programming is an answer.

**Reason2** : If you are an application developer, there are two inconvenient truths:

1. Modern applications are inherently concurrent.

2. Writing concurrent programs that are correct is difficult.

In the domain of mobile or desktop applications, parallel execution allows for responsive user interfaces because we can move computations into the background while the UI responds to ongoing user interactions. Code must execute concurrently to not stray from this fundamental requirement. Writing such programs is diffcult because on mobile they are typically written in imperative languages like C or Java. Writing concurrent code in imperative languages is difficult because code is written in terms of interweaved, temporal instructions that move objects or data structures from one state to another. This imperative style of programming inherently produces side effects. It presents several problems when running instructions in parallel, such as race conditions when writing to a shared resource.

**1. Principle of Reactive programming:**

Reactive programming is an extension of the [Observer software design](https://en.wikipedia.org/wiki/Observer_pattern) pattern, where an object has a list of Observers that are dependent on it, and these Observers are notified by the object whenever it’s state changes.

There are two basic and very important items in reactive programming, [**Observables**](http://reactivex.io/RxJava/javadoc/rx/Observable.html) and [**Observers**](http://reactivex.io/RxJava/javadoc/rx/Observer.html). Observables publish values, while Observers subscribe to Observables, watching them and reacting when an Observable publishes a value.

In simpler terms:

* An Observable performs some action, and publishes the result.(eg: your network operations, some complex calculations)
* An Observer waits and watches the Observable, and reacts whenever the Observable publishes results.(eg: updates the UI)

A class that implements the Observer interface must provide methods for each of the three changes above:

1. An onNext() method that the Observable calls whenever it wishes to publish a new value
2. An onError() method that’s called exactly once, when an error occurs on the Observable.
3. An onCompleted() method that’s called exactly once, when the Observable completes execution.

So an Observable that has an Observer subscribed to it will call the Observer’s onNext() zero or more times, as long as it has values to publish, and terminates by either calling onError() or onCompleted().

RX java setup:

add these lins in build.gradle file.

*// Below lines we have added for reactive programming dependancy*compile **'io.reactivex:rxandroid:1.2.1'**compile **'io.reactivex:rxjava:1.1.9'**

### Operators

The Observable class has many static methods, called **operators**, to create Observable objects. eg: just, form, map, create.

### Side Effect Methods

Side effect methods do not affect your stream in itself. Instead they are invoked when certain events occur to allow you to react to those events.

For example: if you’re interested in doing something outside of your Subscriber‘s callbacks whenever some error occurs, you would use the doOnError() method and pass to it the [functional interface](http://www.lambdafaq.org/what-is-a-functional-interface/) to be used whenever an error occurs:

someObservable

.doOnError(new Action1() {

@Override

public void call(Throwable t) {

// use this callback to clean up resources,

// log the event or or report the

// problem to the user

}

})

//…

### What Are They Useful For?

Now since they do not change the stream of items there must be other uses for them. I present here three examples of what you can achieve using these methods:

* Use doOnNext() for debugging/ save/cache network results
* Use doOnError() to clean the resources

So let’s see these examples in detail.

#### Use doOnNext() for debugging

Here’s an example of this:

Observable someObservable = Observable

.from(Arrays.asList(new Integer[]{2, 3, 5, 7, 11}))

.doOnNext(System.out::println)

.filter(prime -> prime % 2 == 0)

.doOnNext(System.out::println)

.count()

.doOnNext(System.out::println)

.map(number -> String.format(“Contains %d elements”, number));

Subscription subscription = o.subscribe(

System.out::println,

System.out::println,

() -> System.out.println(“Completed!”));

The doOnError() and doOnCompleted() methods can also be useful for debugging the state of your pipeline.

**Note:** If you’re using RxJava while developing for Android please have a look at the [Frodo](https://github.com/android10/frodo) and Fernando Ceja’s post explaining about the [motivation for and usage of Frodo](http://fernandocejas.com/2015/11/05/debugging-rxjava-on-android/). With Frodo you can use annotations to debug your Observables and Subscribers.

The shown way of using doOnNext() and doOnError() does not change much of the system state – apart from bloating your log and slowing everything down.

But there are other uses for these operators. And in those cases you use those methods to actually change the state of your system. Let’s have a look at them.

This method is especially useful if you query your remote resource as a result of potentially recurring UI events.

#### Use doOnNext() to save/cache network results

If at some point in your chain you make network calls, you could use doOnNext() to store the incoming results to your local database or put them in some cache.

**Schedulers**

There are several types of Schedulers available in the RxJava framework, but the tricky part is to choose the right Scheduler for the right kind of job. Your task will never run optimally if you don’t pick the right one. So, let’s try and understand each of them -

#### Schedulers.io()

**It is backed by an unbounded thread pool** and is used for non-CPU intensive I/O work like accessing the file system, performing network calls, accessing the database, etc. This Scheduler is uncapped and the size of its thread pool can grow as needed.

#### Schedulers.computation()

This Scheduler is used for performing CPU-intensive work like processing large data sets, image handling, etc. **It is backed by a bounded thread pool**with size up to the number of processors available.

As this Scheduler is suitable for only CPU intensive tasks, we want to limit the number of threads so that they don’t fight among each other over CPU time and starve themselves.

#### Schedulers.newThread()

This Scheduler creates a **completely new thread to perform a unit of work every time** it is used. It doesn’t benefit itself by making use of any thread pool. Threads are expensive to create and tear down, so you should be pretty careful about not abusing excessive thread spawning leading to severe system slowdowns and out of memory errors.

Ideally, you would use this Scheduler quite rarely, mostly for kicking off long-running, isolated units of work in a completely separate thread.

#### Schedulers.single()

This Scheduler is newly introduced in RxJava 2 and is **backed by a single thread which can only be used to perform tasks** in a sequential manner. This can be highly useful when you have a set of background jobs to perform from different places in your app but can’t afford to execute more than one at a time.

#### Schedulers.from(Executor executor)

You can use this to create a custom Scheduler backed by your own Executor. There can be several use cases where you might need to create a custom Scheduler to perform specific tasks for your app that demands custom threading logic.

Suppose, you want to limit the number of parallel network calls happening in your app, then you can create a custom Scheduler with an executor of fixed thread pool size, Scheduler.from(Executors.newFixedThreadPool(n)) and use it on all network-related Observables in your code.

#### AndroidSchedulers.mainThread()

This is a special Scheduler which is not available in the core RxJava library. You need to use the [RxAndroid](https://github.com/ReactiveX/RxAndroid" \t "_blank) extension library to make use of it. This **scheduler is specifically useful for Android apps to perform UI based tasks** in the main thread of the application.

By default, it enqueues tasks in the looper associated with the main thread of the application, but there are other variations of it that allows us to use any Looper we want via APIs like this, AndroidSchedulers.from(Looper looper).

**Note**: Be careful while using Schedulers backed by unbounded thread pools like Schedulers.io(), as there always lies a risk of infinitely growing the thread pool and flooding the system with too many threads.

**References:**

**RX java basics:**

<https://www.androidauthority.com/reactive-programming-with-rxandroid-711104/>

**Problems with asynctask and solution given by Rx java**

<https://stablekernel.com/replace-asynctask-and-asynctaskloader-with-rx-observable-rxjava-android-patterns/>

<https://mttkay.github.io/blog/2013/08/25/functional-reactive-programming-on-android-with-rxjava/>

**Subjects:**

<https://blog.mindorks.com/understanding-rxjava-subject-publish-replay-behavior-and-async-subject-224d663d452f>

**Different types of filter:**

<https://github.com/ReactiveX/RxJava/wiki/Filtering-Observables>

Imp Link:

<https://medium.com/@Vijay_S/using-rxandroid-to-fix-callback-hell-feed172118f5>

<https://android.jlelse.eu/what-should-you-know-about-rx-operators-54716a16b310>

All Operators

<http://reactivex.io/documentation/operators.html>

http://jdam.cd/async-android/