MENU

Grokking RxJava, Part 1: The Basics

15 SEPTEMBER 2014 on rxjava

RxJava is the new hotness amongst Android developers these days. The only problem is that it can be difficult to approach initially. Functional Reactive Programming is hard when you come from an imperative world, but once you understand it, it's so awesome!

I'm here to try to give you a flavor of RxJava. The goal of this ~~three~~ four-part series is to get your foot in the door. I'm not going to try to explain everything (nor could I). I just want you to become interested in RxJava and how it works.

The Basics

The basic building blocks of reactive code are `Observables` and `Subscribers`[1]. An `Observable` emits items; a `Subscriber` consumes those items.

There is a pattern to how items are emitted. An `Observable` may emit any number of items (including zero items), then it terminates either by successfully completing, or due to an error. For each `Subscriber` it has, an `Observable` calls `Subscriber.onNext()` any number of times, followed by either `Subscriber.onComplete()` or `Subscriber.onError()`.

This looks a lot like your standard observer pattern, but it differs in one key way - `Observables` often don't start emitting items until someone explicitly subscribes to them[2]. In other words: if no one is there to listen, the tree won't fall in the forest.

Hello, World!

Let's see this framework in action with a concrete example. First, let's create a basic `Observable`:

```
Observable<String> myObservable = Observable.create(
    new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> sub)
{
            sub.onNext("Hello, world!");
            sub.onCompleted();
        }
    }
);
```

Our `Observable` emits "Hello, world!" then completes. Now let's create a `Subscriber` to consume the data:

```
Subscriber<String> mySubscriber = new Subscriber<String>
() {
    @Override
    public void onNext(String s) { System.out.println(s);
}
```

```
    @Override
    public void onCompleted() { }

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

All this does is print each String emitted by the `Observable`.

Now that we've got `myObservable` and `mySubscriber` we can hook them up to each other using `subscribe()`:

```
myObservable.subscribe(mySubscriber);
// Outputs "Hello, world!"
```

When the subscription is made, `myObservable` calls the subscriber's `onNext()` and `onComplete()` methods. As a result, `mySubscriber` outputs "Hello, world!" then terminates.

Simpler Code

That's a lot of boilerplate code just to say "Hello, world!" That's because I took the verbose route so you could see exactly what's happening. There are lots of shortcuts provided in RxJava to make coding easier.

First, let's simplify our `Observable`. RxJava has multiple built-in `Observable` creation methods for common tasks. In this case,

`Observable.just()` emits a single item then completes, just like our code above[3]:

```
Observable<String> myObservable =
    Observable.just("Hello, world!");
```

Next, let's handle that unnecessarily verbose `Subscriber`. We don't care about `onCompleted()` nor `onError()`, so instead we can use a simpler class to define what to do during `onNext()`:

```
Action1<String> onNextAction = new Action1<String>() {
    @Override
    public void call(String s) {
        System.out.println(s);
    }
};
```

Actions can define each part of a `Subscriber`. `Observable.subscribe()` can handle one, two or three `Action` parameters that take the place of `onNext()`, `onError()`, and `onComplete()`. Replicating our `Subscriber` from before looks like this:

```
myObservable.subscribe(onNextAction, onErrorAction,
onCompleteAction);
```

However, we only need the first parameter, because we're ignoring `onError()` and `onComplete()`:

```
myObservable.subscribe(onNextAction);
// Outputs "Hello, world!"
```

Now, let's get rid of those variables by just chaining the method calls together:

```
Observable.just("Hello, world!")
    .subscribe(new Action1<String>() {
        @Override
        public void call(String s) {
              System.out.println(s);
        }
    });
```

Finally, let's use Java 8 lambdas to get rid of that ugly `Action1` code.

```
Observable.just("Hello, world!")
    .subscribe(s -> System.out.println(s));
```

If you're on Android (and thus can't use Java 8), I *highly* recommend using retrolambda; it will cut down on the verbosity of your code immensely.

Transformation

Let's spice things up.

Suppose I want to append my signature to the "Hello, world!" output. One possibility would be to change the `Observable`:

```
Observable.just("Hello, world! -Dan")
    .subscribe(s -> System.out.println(s));
```

This works if you have control over your `Observable`, but there's no guarantee that will be the case - what if you're using someone else's library? Another potential problem: what if I use my `Observable` in multiple places but only *sometimes* want to add the signature?

How about we try modifying our `Subscriber` instead:

```
Observable.just("Hello, world!")
    .subscribe(s -> System.out.println(s + " -Dan"));
```

This answer is also unsatisfactory, but for different reasons: I want my `Subscribers` to be as lightweight as possible because I might be running them on the main thread. On a more conceptual level, `Subscribers` are supposed to be the thing that *reacts*, not the thing that *mutates*.

Wouldn't it be cool if I could transform "Hello, world!" with some intermediary step?

Introducing Operators

Here's how we're going to solve the item transformation problems: with operators. Operators can be used in between the source `Observable` and the ultimate `Subscriber` to manipulate emitted items. RxJava comes with a huge collection of operators, but its best to focus on just a handful at first.

For this situation, the `map()` operator can be used to transform one emitted item into another:

```
Observable.just("Hello, world!")
    .map(new Func1<String, String>() {
        @Override
        public String call(String s) {
            return s + " -Dan";
        }
    })
    .subscribe(s -> System.out.println(s));
```

Again, we can simplify this by using lambdas:

```
Observable.just("Hello, world!")
    .map(s -> s + " -Dan")
    .subscribe(s -> System.out.println(s));
```

Pretty cool, eh? Our `map()` operator is basically an `Observable` that transforms an item. We can chain as many `map()` calls as we want together, polishing the data into a perfect, consumable form for our end `Subscriber`.

More on map()

Here's an interesting aspect of `map()`: it does not have to emit items of the same type as the source `Observable`!

Suppose my `Subscriber` is not interested in outputting the original text, but instead wants to output the hash of the text:

```
Observable.just("Hello, world!")
    .map(new Func1<String, Integer>() {
        @Override
        public Integer call(String s) {
            return s.hashCode();
        }
    })
    .subscribe(i ->
System.out.println(Integer.toString(i)));
```

Interesting - we started with a String but our `Subscriber` receives an Integer. Again, we can use lambdas to shorten this code:

```
Observable.just("Hello, world!")
    .map(s -> s.hashCode())
    .subscribe(i ->
System.out.println(Integer.toString(i)));
```

Like I said before, we want our `Subscriber` to do as little as possible. Let's throw in another `map()` to convert our hash back into a String:

```
Observable.just("Hello, world!")
    .map(s -> s.hashCode())
    .map(i -> Integer.toString(i))
    .subscribe(s -> System.out.println(s));
```

Would you look at that - our `Observable` and `Subscriber` are back to their original code! We just added some transformational steps in between. We could even add my signature transformation back in as well:

```
Observable.just("Hello, world!")
    .map(s -> s + " -Dan")
    .map(s -> s.hashCode())
    .map(i -> Integer.toString(i))
    .subscribe(s -> System.out.println(s));
```

So What?

At this point you might be thinking "that's a lot of fancy footwork for some simple code." True enough; it's a simple example. But there's two ideas you should take away:

Key idea #1: `Observable` and `Subscriber` can do anything.

Let your imagination run wild! Anything is possible.

Your `Observable` could be a database query, the `Subscriber` taking the results and displaying them on the screen. Your `Observable` could be a click on the screen, the `Subscriber` reacting to it. Your `Observable` could be a stream of bytes read from the internet, the `Subscriber` could write it to the disk.

It's a general framework that can handle just about any problem.

Key idea #2: The `Observable` and `Subscriber` are independent of the transformational steps in between them.

I can stick as many `map()` calls as I want in between the original source `Observable` and its ultimate `Subscriber`. The system is highly **composable**:

it is easy to manipulate the data. As long as the operators work with the correct input/output data I could make a chain that goes on forever[4].

Combine our two key ideas and you can see a system with a lot of potential. At this point, though, we only have a single operator, `map()`, which severely limits our capabilities. In part 2 we'll take a dip into the large pool of operators available to you when using RxJava.

Continue onwards to part 2

---

[1] The smallest building block is actually an `Observer`, but in practice you are most often using `Subscriber` because that's how you hook up to `Observables`.

[2] The term used is "hot" vs. "cold" `Observables`. A hot `Observable` emits items all the time, even when no one is listening. A cold `Observable` only emits items when it has a subscriber (and is what I'm using in all my examples). This distinction isn't that important to initially learning RxJava.

[3] Strictly speaking, `Observable.just()` isn't *exactly* the same as our initial code, but I won't get to *why* until part 3.

[4] Okay, not really, since I'll hit the bounds of the machine at some point, but you get the idea.

Dan Lew

Share this post

Read more posts by this author.

Minneapolis *http://blog.danlew.net/about/*

98 Comments Dan Lew Codes 1 Login

Recommend 67 Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

anticafe • 3 years ago

I've read many get started RxJava tutorials but yours is one of the easiest to understand. Thanks so much.

50 • Reply • Share ›

Kiran Rao ➚ anticafe • 3 years ago

I second **@anticafe** s opinion here. I have read several articles, gone through videos, presentations and what not on RxJava. While in most cases I could see the benefits of RxJava, I always had trouble with the first step - engine ignition if I may. Thanks to your approach of taking the absolute basic concepts first and then building on them, it is now all crystal clear to me. Thanks!

9 • Reply • Share ›

Javen ck • 3 years ago

flatmap request sameurl with diff param but the one exception the after exception subscribe can't reciever observer's message

14 • Reply • Share ›

clarkbattle • a year ago

This is a great tutorial. The best Ive read on the topic. The only thing I would change about it is that a good tutorial should only introduce one topic and not rely on others. Showing that it is possible to use lambdas is interesting but if you dont understand them it really detracts from the clarity of understanding reactive programming. You could have put the lambdas in a side note and just used method calls to shorten the boilerplate code. For example you could simply use a method that takes a string and returns an Action1 that prints it. That would accomplish the same goal of shortening the code without requiring the reader to understand two new programming methodologies at the same time. Its a common mistake, which is the reason why most other tutorials are so hard to understand. Focus on explaining one thing and one thing only, resisting the urge to throw in other paradigms, regardless of how useful they may be in real programming. Great job otherwise. Thanks.

6 • Reply • Share ›

Aaron Ronoh • 2 years ago

You broke it down to understandable bits. Thanks

4 ^ | v • Reply • Share ›

Zain Ul Abideen • 5 months ago

I'm glad I found this post here. Very well explained.

1 ^ | v • Reply • Share ›

kgoralski • 2 years ago

Hi Dan, your guide was very helpful. I made a very basic code snippet based on your tutorial:
https://krzysztofgoralski.w...

1 ^ | v • Reply • Share ›

lekkie omotayo • 3 years ago

@Dan Lew

I dont use Java8 and I am not developing on android. It appears I cant use the lambda neither can I use the retrolambda. I tried to use the verbose option, but I am having issues with cast from String to Integer. Any advice on how to proceed?

1 ^ | v • Reply • Share ›

Alex Felipe ➚ lekkie omotayo • a year ago

lekkie, currenty exists this project for use lambda in Android
https://github.com/evant/gr...

^ | v • Reply • Share ›

Renso ➚ lekkie omotayo • 3 years ago

I'm also having the same issues, I would love to see this whole tutorial in the verbose route and after I understand the whole concept I would spend some time trying to make code smaller, but is just my opinion. Still you do a great job with this, Thanks a lot.

^ | v • Reply • Share ›

Dan Lew Mod ➚ lekkie omotayo • 3 years ago

I think I'd need to see a bit more code to help out.

^ | v • Reply • Share ›

Dominic Fui Dodzi-Nusenu • 3 years ago

Love this article

1 ^ | v • Reply • Share ›

Sait Sami Kocataş • 3 years ago

Great article, thanks

1 ^ | v • Reply • Share ›

Tomasz Rykala • 3 years ago

This is the proper introduction and explanation of benefits of RxJava, that I've seen yet. Getting to see the benefits of adding retrolambda is a bonus. Thank you, finally got it.

1 ^ | v • Reply • Share ›

This comment was deleted.

ber4444 ➚ Guest • 3 years ago

Even Square switched to Rx from Otto, see https://github.com/JakeWhar...

^ | v • Reply • Share ›

Dan Lew Mod ➚ ber4444 • 3 years ago

Event buses and functional reactive programming can serve similar functions but FRP can do a lot more ultimately. Event buses are simple observers, FRP is based on the observer pattern but has additional functionality beyond that. There's no reason you couldn't use both, too - have an event bus broadcast events globally, which are picked up by an RxJava chain to do some operations locally.

u2020 uses RxJava, so I'm not sure what you're talking about... It's a sample app anyways, not really indicative of what Square uses but rather it's a presentation of a many open source libs working together.

Here's the class where u2020 features RxJava:
https://github.com/JakeWhar...

1 ^ | v • Reply • Share ›

ber4444 ➚ Dan Lew • 3 years ago

read my comment again :)

^ | v • Reply • Share ›

Dan Lew Mod ➚ ber4444 • 3 years ago

I have done so and still don't get your point. You're going to have to use your words.

3 ^ | v • Reply • Share ›

ber4444 ➚ Dan Lew • 3 years ago

What I meant is that you understood my comment to mean the exact opposite, even after your second read :) Anyway, look at this about Otto vs Rx:

Alexander Blom @blomalexander Oct 3, 2013
Replying to @JakeWharton
@JakeWharton How is RxJava different from composable futures? Except the stream stuff

which I've never seen used.

Jake Wharton
@JakeWharton

@blomalexander More importantly, it kills just about all use of Otto and gives you request joining for free.

11:37 PM - Oct 3, 2013

Reply • Share ›

Dan Lew Mod ber4444 • 3 years ago

Ahh, okay. I don't know if I agree that Rx replaces Otto TBH. Otto is simple and all you need sometimes.

2 Reply • Share ›

김영진 • 2 months ago

Good morning
I want to translate my blog and upload it to my site.
Would it be okay?
The post is from Grokking RxJava 1 to 4.

Reply • Share ›

hardyeats • 4 months ago

Hello, Dan.
I've translated your article into my mother tongue and modified the example code to be compatible with RxJava 2.
https://hardyeats.github.io...
I apologize for doing this without your permission.
I will delete this post immediately if you want.

Reply • Share ›

dali high • 7 months ago

AWESOME

Reply • Share ›

anurag soni • 8 months ago

Awesome initial start for RxJava , I used to avoid use of it in projects but now this tutorial gives confidence for my development using RxJava.
Thanks a ton for writing it with so simplicity.
I will go through next posts also and thanking you for them in advance Dan.

Reply • Share ›

Àlex Amenós • 9 months ago

Thanks Dan, very instructive and helpful to start.

Reply • Share ›

Michael Angelo Rivera • 10 months ago

Any update for the 2.0 version?

Reply • Share ›

Dan Lew Mod → Michael Angelo Rivera • 10 months ago

Maybe someday? The fundamental ideas in the articles all still apply to RxJava 2.0.

Reply • Share ›

Jim Andreas → Dan Lew • 7 months ago

Awesome article! Thanks Dan. Regarding 2.0 - here is a code snippet with the small details changed. This snip runs in IntelliJ Idea if you pull in the

```
    io.reactivex.rxjava2:rxjava:2.1.0
```

library from maven in your Project Structure dialog. Notes for 2.0:

Action1 is now Consumer, while call() is now subscribe(). Note that for some unknown reason, the "String" type for Consumer is changed to lower case when this is posted. Change it back to "String" (upper case) when you paste it into Idea. :-)

```
public class RxJava {
    @Test
    public void doTest() {
        Observable.just("Hello, world!")
                .subscribe(new Consumer<string>() {
                    @Override
                    public void accept(String s) {
                        System.out.println(s);
                    }
                });
    }
}
```

Reply • Share ›

Ho Yin • a year ago

Very great article. Agree, it is the easiest to understand the whole idea of RxJava

Reply • Share ›

Mark • a year ago

Excellent tutorial. Very clear with easy to understand working examples. I'm looking forward to going through your next two articles!

Reply • Share ›

jamesbluecrow • a year ago

Amazing post! Thanks for taking the time to share what you've learnt.

Reply • Share ›

Mez Pahlan • a year ago

Cheers Dan! Easiest to understand intro I've seen!

Reply • Share ›

Alex Felipe • a year ago

Dan, thanks for introduction, i'm testing this framework and i want to learn for help pt-br community :)

Reply • Share ›

Mahdi Pishguy • a year ago

how can i use that on Android service such as service or intentService

Reply • Share ›

Dharmar Gurusamy • a year ago

Tutorial is very great. I would like test the sample programs you have given in IntelliJ IDE.
Could by you please inform me as what all I should import so that I can run the program.
Now I am getting red mark all aver. Thank You.

Reply • Share ›

eoin • a year ago

rxjava is really powerful. i really need to start using it. great article by the way

Reply • Share ›

Neon Warge • a year ago

I really on the other side of the spectrum and I felt sad and frustrated. I really can't understand this reactive programming. I can't really see the point. Maybe that time will come soon, I need to spend more time with this. As of now, I am intrigued, not fascinated. I am still trying to wrap my head around its practicality.

Very good article though.

Reply • Share ›

Aditya Pise • a year ago

Thank you! I saw almost every video on youtube on RxJava but didn't understand it. Your blog is one of the best resource on INTERNET for RxJava.

Reply • Share ›

Trinh Duy Hung • a year ago

Yes Dan Lew you saved my day. Thank your very much.

Reply • Share ›

Karthik Kolanji • 2 years ago

Dude your tutorial is dead simple to get started . Thanks a lot.

Reply • Share ›

writing services reviews • 2 years ago

There would surely be a lot of people who will be going to learn something from this that will effectively guide them up and would effectively promote a lot of techniques on their programming activities

Reply • Share ›

ryanhoo • 2 years ago

Nice work

Reply • Share ›

Gondole • 2 years ago

nice

Reply • Share ›

Ahmed marzouk • 2 years ago

Thanks, its very good easy to be understood :D

Reply • Share ›

Aidan Mack • 2 years ago

So can I have my observable in one class and and my subscriber in another? If subscriber is in another class and I want to use a map with my observable how would I get reference to it to add one?

Reply • Share ›

julien • 2 years ago

This is by far the best resource/tutorial on RxJava I've seen, really helped me "grok" the subject, thanks a lot.
I don't know if it's just me but Observable.subscribe() does not feel very natural, I would have thought the subscriber should have that method no? Or perhaps Observable.addSubscriber() anyway just semantic I guess

Reply • Share ›

Dan Lew Mod → julien • 2 years ago

Think of it as subscribing TO the Observable.

Reply • Share ›

roberto_artiles • 2 years ago

Hey Dan,

Thanks for the extensive 4 articles on rx-java. They were very helpful.
The only thing I'm struggling to understand with all these chained operations: is it

normal to have operators which use variables/arguments/fields from the outer scope. For me it feels really fishy. But the only way I can think of to pass multiple params between operators, is by using some kind of tupling mechanism (e.g. http://www.javatuples.org/), or wrapping them in my own class. Am I missing something?

Example to illustrate the possible case:
I need to process and store in database "id", "name" and "photo" by chaining operators starting from Observable.just(id). In the first flatMap I want to alter the name. In the second flatMap I want to write the photo on disk. And in the third flatMap I want to put in DB all 3 values, including the altered name + photo's path on the disk . So, as you can

READ THIS NEXT

Grokking RxJava, Part 2: Operator, Operator

In part 1 I went over the basic structure of RxJava, as well as introducing you to the map(…

YOU MIGHT ENJOY

What Should You Localize?

Localization is an important topic to me; there's no faster way to expand your audience than include the entire…