RxJava

**Grokking RxJava, Part 1: The Basics**

15 SEPTEMBER 2014 on [rxjava](http://blog.danlew.net/tag/rxjava/)

[RxJava](https://github.com/ReactiveX/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  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 Subscribers1. 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](http://en.wikipedia.org/wiki/Observer_pattern), but it differs in one key way - Observables often don't start emitting items until someone explicitly subscribes to them2. 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 Observablecreation methods for common tasks. In this case, Observable.just() emits a single item then completes, just like our code above3:

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](https://github.com/evant/gradle-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](http://www.zombo.com/).

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 forever4.

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.

**Grokking RxJava, Part 2: Operator, Operator**

22 SEPTEMBER 2014 on [rxjava](http://blog.danlew.net/tag/rxjava/)

In [part 1](http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/) I went over the basic structure of RxJava, as well as introducing you to the map() operator. However, I can understand if you're still not compelled to use RxJava - you don't have much to work with yet. But that will change quickly - a big part of the power of RxJava is in all of the operators included in the framework.

Let's go through an example to introduce you to more operators.

**The Setup**

Suppose I have this method available:

// Returns a List of website URLs based on a text search

Observable<List<String>> query(String text);

I want to make a robust system for searching text and displaying the results. Given what we know from the last article, this is what one might come up with:

query("Hello, world!")

.subscribe(urls -> {

for (String url : urls) {

System.out.println(url);

}

});

This answer is highly unsatisfactory because I lose the ability to transform the data stream. If I wanted to modify *each* URL, I'd have to do it all in the Subscriber. We're tossing all our cool map() tricks out the window!

I could create a map() from urls -> urls, but then *every* map() call would have a for-each loop inside of it - ouch.

**A Glimmer of Hope**

There is a method, Observable.from(), that takes a collection of items and emits each them one at a time:

Observable.from("url1", "url2", "url3")

.subscribe(url -> System.out.println(url));

That looks like it could help, let's see what happens:

query("Hello, world!")

.subscribe(urls -> {

Observable.from(urls)

.subscribe(url -> System.out.println(url));

});

I've gotten rid of the for-each loop, but the resulting code is a *mess*. I've got multiple, nested subscriptions now! Besides being ugly and hard to modify, it also breaks some critical as-yet undiscovered features of RxJava1. Ugh.

**A Better Way**

Hold your breath as you view your savior: flatMap().

Observable.flatMap() takes the emissions of one Observable and returns the emissions of another Observable to take its place. It's the ol' switcheroo: you thought you were getting one stream of items but instead you get another. Here's how it solves this problem:

query("Hello, world!")

.flatMap(new Func1<List<String>, Observable<String>>() {

@Override

public Observable<String> call(List<String> urls) {

return Observable.from(urls);

}

})

.subscribe(url -> System.out.println(url));

I'm showing the full function just so you can see exactly what happened, but simplified with lambdas it looks awesome:

query("Hello, world!")

.flatMap(urls -> Observable.from(urls))

.subscribe(url -> System.out.println(url));

flatMap() is weird, right? Why is it returning *another* Observable? The key concept here is that the new Observable returned is what the Subscriber sees. It doesn't receive a List<String> - it gets a series of individual Strings as returned by Observable.from().

For the record, this part was the hardest for me to understand, but once I had the "aha" moment a lot of RxJava clicked.

**It Gets Even Better**

I can't emphasize this idea enough: flatMap() can return *any* Observable it wants.

Suppose I've got a second method available:

// Returns the title of a website, or null if 404

Observable<String> getTitle(String URL);

Instead of printing the URLs, now I want to print the title of each website received. But there's a few issues: my method only works on a single URL at a time, and it doesn't return a String, it returns an Observable that emits the String.

With flatMap(), solving this problem is easy; after splitting the list of URLs into individual items, I can use getTitle() in flatMap() for each url before it reaches the Subscriber:

query("Hello, world!")

.flatMap(urls -> Observable.from(urls))

.flatMap(new Func1<String, Observable<String>>() {

@Override

public Observable<String> call(String url) {

return getTitle(url);

}

})

.subscribe(title -> System.out.println(title));

And once more, simplified via lambdas:

query("Hello, world!")

.flatMap(urls -> Observable.from(urls))

.flatMap(url -> getTitle(url))

.subscribe(title -> System.out.println(title));

Crazy, right? I'm composing multiple independent methods returning Observables together! How cool is that!

Not only that, but notice how I'm combining two API calls into a single chain. We could do it for any number of API calls. You know how much of a pain in the ass it is to keep all your API calls synced, having to link their callbacks together before presenting the data? We've skipped the trip to callback hell; all that same logic is now encased in this short reactive call2.

**Operators Galore**

We've only looked at two operators so far, but there are so many more! How else can we improve our code?

getTitle() returns null if the URL 404s. We don't want to output "null"; it turns out we can filter them out!

query("Hello, world!")

.flatMap(urls -> Observable.from(urls))

.flatMap(url -> getTitle(url))

.filter(title -> title != null)

.subscribe(title -> System.out.println(title));

filter() emits the same item it received, but only if it passes the boolean check.

And now we want to only show 5 results at most:

query("Hello, world!")

.flatMap(urls -> Observable.from(urls))

.flatMap(url -> getTitle(url))

.filter(title -> title != null)

.take(5)

.subscribe(title -> System.out.println(title));

take() emits, at most, the number of items specified. (If there are fewer than 5 titles it'll just stop early.)

Now we want to save each title to disk along the way:

query("Hello, world!")

.flatMap(urls -> Observable.from(urls))

.flatMap(url -> getTitle(url))

.filter(title -> title != null)

.take(5)

.doOnNext(title -> saveTitle(title))

.subscribe(title -> System.out.println(title));

doOnNext() allows us to add extra behavior each time an item is emitted, in this case saving the title.

Look at how easy it is to manipulate the stream of data. You can keep adding more and more ingredients to your recipe and not mess anything up.

[RxJava comes with a **ton** of operators](https://github.com/ReactiveX/RxJava/wiki/Alphabetical-List-of-Observable-Operators). It is intimidating how many operators there are, but it's worth reviewing so you know what's available. It will take a while to internalize the operators but you'll have true power at your fingertips once you do.

On top of all that's provided, you can even write your own custom operators! That's outside the scope of this article, but basically, if you can think it, you can do it3.

**So What?**

Alright, so you're a hard sell. You're a skeptic. Why should you care about all these operators?

**Key idea #3: Operators let you do anything to the stream of data.**

[The only limit is yourself](http://www.zombo.com/).

You can setup complex logic using nothing but chains of simple operators. It breaks down your code into composable bits and pieces. That's functional reactive programming. The more you use it, the more it changes the way you think about programs.

Plus, think about how simple our data was to consume once transformed. By the end of our example we were doing two API calls, manipulating the data, then saving it to disk. But our Subscriber doesn't know that; it just thinks it's consuming a simple Observable<String>. Encapsulation makes coding easier!

In part 3 we'll cover some of the other cool features of RxJava that aren't as directly involved with manipulating data, like error handling and concurrency.

# Grokking RxJava, Part 3: Reactive with Benefits

30 SEPTEMBER 2014 on [rxjava](http://blog.danlew.net/tag/rxjava/)

In [part 1](http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/), I went over the basic structure of RxJava. In [part 2](http://blog.danlew.net/2014/09/22/grokking-rxjava-part-2/), I showed you how powerful operators could be. But maybe you're still not sold; there isn't quite enough there yet to convince you. Here's some of the other benefits that come along with the RxJava framework which should seal the deal.

# Error Handling

Up until this point, we've largely been ignoring onComplete() and onError(). They mark when an Observable is going to stop emitting items and the reason for why (either a successful completion, or an unrecoverable error).

Our original Subscriber had the capability to listen to onComplete() and onError(). Let's actually do something with them:

Observable.just("Hello, world!")

.map(s -> potentialException(s))

.map(s -> anotherPotentialException(s))

.subscribe(new Subscriber<String>() {

@Override

public void onNext(String s) { System.out.println(s); }

@Override

public void onCompleted() { System.out.println("Completed!"); }

@Override

public void onError(Throwable e) { System.out.println("Ouch!"); }

});

Let's say potentialException() and anotherPotentialException() both have the possibility of throwing Exceptions. Every Observable ends with either a single call to onCompleted() or onError(). As such, the output of the program will either be a String followed by "Completed!" or it will just be "Ouch!" (because an Exception is thrown).

There's a few takeaways from this pattern:

1. **onError() is called if an Exception is thrown at any time.**

This makes error handling much simpler. I can just handle every error at the end in a single function.

1. **The operators don't have to handle the Exception.**

You can leave it up to the Subscriber to determine how to handle issues with any part of the Observable chain because Exceptions skip ahead to onError().

1. **You know when the Subscriber has finished receiving items.**

Knowing when a task is done helps the flow of your code. (Though it is possible that an Observable may never complete.)

I find this pattern a lot easier than traditional error handling. With callbacks, you have to handle errors in each callback. Not only does that lead to repetitious code, but it also means that each callback must know how to handle errors, meaning your callback code is tightly coupled to the caller.

With RxJava's pattern, your Observable doesn't even have to know what to do with errors! Nor do any of your operators have to handle error states - they'll be skipped in cases of critical failure. You can leave all your error handling to the Subscriber.

# Schedulers

You've got an Android app that makes a network request. That could take a long time, so you load it in another thread. Suddenly, you've got problems!

Multi-threaded Android applications are difficult because you have to make sure to run the right code on the right thread; mess up and your app can crash. The classic exception occurs when you try to modify a View off of the main thread.

In RxJava, you can tell your Observer code which thread to run on using subscribeOn(), and which thread your Subscriber should run on using observeOn():

myObservableServices.retrieveImage(url)

.subscribeOn(Schedulers.io())

.observeOn(AndroidSchedulers.mainThread())

.subscribe(bitmap -> myImageView.setImageBitmap(bitmap));

How simple is that? Everything that runs before my Subscriber runs on an I/O thread. Then in the end, my View manipulation happens on the main thread1.

The great part about this is that I can attach subscribeOn() and observeOn() to any Observable! They're just operators! I don't have to worry about what the Observable or its previous operators are doing; I can just stick this at the end for easy threading2.

With an AsyncTask or the like, I have to design my code around which parts of the code I want to run concurrently. With RxJava, my code stays the same - it's just got a touch of concurrency added on.

# Subscriptions

There's something I've been hiding from you. When you call Observable.subscribe(), it returns a Subscription. This represents the link between your Observable and your Subscriber:

Subscription subscription = Observable.just("Hello, World!")

.subscribe(s -> System.out.println(s));

You can use this Subscription to sever the link later on:

subscription.unsubscribe();

System.out.println("Unsubscribed=" + subscription.isUnsubscribed());

// Outputs "Unsubscribed=true"

What's nice about how RxJava handles unsubscribing is that it stops the chain. If you've got a complex chain of operators, using unsubscribe will terminate wherever it is currently executing code3. No unnecessary work needs to be done!

# Conclusion

Keep in mind that these articles are an introduction to RxJava. There's a lot more to learn than what I presented and it's not all sunshine and daisies (for example, read up on [backpressure](https://github.com/ReactiveX/RxJava/wiki/Backpressure)). Nor would I use reactive code for everything - I reserve it for the more complex parts of the code that I want to break into simpler logic.

Originally, I had planned for this post to be the conclusion of the series, but a common request has been for some practical RxJava examples in Android, so you can now [continue onwards to part 4](http://blog.danlew.net/2014/10/08/grokking-rxjava-part-4/). I hope that this introduction is enough to get you started on a fun framework. If you want to learn more, I suggest reading [the official RxJava wiki](https://github.com/ReactiveX/RxJava/wiki). And remember: [the infinite is possible](http://zombo.com/).

**Grokking RxJava, Part 4: Reactive Android**

08 OCTOBER 2014 on [android](http://blog.danlew.net/tag/android/), [rxjava](http://blog.danlew.net/tag/rxjava/)

In parts [1](http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/), [2](http://blog.danlew.net/2014/09/22/grokking-rxjava-part-2/), and [3](http://blog.danlew.net/2014/09/30/grokking-rxjava-part-3/) I covered how RxJava works (in a general sense). But as an Android developer, how do you make it work *for you*? Here is some practical information for Android developers.

**RxAndroid**

[RxAndroid](https://github.com/ReactiveX/RxAndroid) is an extension to RxJava built just for Android. It includes special bindings that will make your life easier.

First, there's AndroidSchedulers which provides schedulers ready-made for Android's threading system. Need to run some code on the UI thread? No problem - just use AndroidSchedulers.mainThread():

retrofitService.getImage(url)

.subscribeOn(Schedulers.io())

.observeOn(AndroidSchedulers.mainThread())

.subscribe(bitmap -> myImageView.setImageBitmap(bitmap));

If you've got your own Handler, you can create a scheduler linked to it with HandlerThreadScheduler1.

Next we have AndroidObservable which provides more facilities for working within the Android lifecycle. There is bindActivity() and bindFragment() which, in addition to automatically using AndroidSchedulers.mainThread() for observing, will also stop emitting items when your Activity or Fragment is finishing (so you don't accidentally try to change state after it is valid to do so).

AndroidObservable.bindActivity(this, retrofitService.getImage(url))

.subscribeOn(Schedulers.io())

.subscribe(bitmap -> myImageView.setImageBitmap(bitmap));

I also like AndroidObservable.fromBroadcast(), which allows you to create an Observable that works like a BroadcastReceiver. Here's a way to be notified whenever network connectivity changes:

IntentFilter filter = new IntentFilter(ConnectivityManager.CONNECTIVITY\_ACTION);

AndroidObservable.fromBroadcast(context, filter)

.subscribe(intent -> handleConnectivityChange(intent));

Finally, there is ViewObservable, which adds a couple bindings for Views. There's ViewObservable.clicks() if you want to get an event each time a View is clicked, or ViewObservable.text() to observe whenever a TextView changes its content.

ViewObservable.clicks(mCardNameEditText, false)

.subscribe(view -> handleClick(view));

**Retrofit**

There's one notable library that supports RxJava: [Retrofit](https://github.com/square/retrofit), a popular REST client for Android. Normally when you define an asynchronous method you add a Callback:

@GET("/user/{id}/photo")

void getUserPhoto(@Path("id") int id, Callback<Photo> cb);

With RxJava installed, you can have it return an Observable instead:

@GET("/user/{id}/photo")

Observable<Photo> getUserPhoto(@Path("id") int id);

Now you can hook into the Observable any way you want; not only will you get your data but you can transform it, too!

Retrofit support for Observable also makes it easy to combine multiple REST calls together. For example, suppose we have one call that gets the photo and a second that gets the metadata. We can zip the results together:

Observable.zip(

service.getUserPhoto(id),

service.getPhotoMetadata(id),

(photo, metadata) -> createPhotoWithData(photo, metadata))

.subscribe(photoWithData -> showPhoto(photoWithData));

I showed a similar example to this in [part 2](http://blog.danlew.net/2014/09/22/grokking-rxjava-part-2/) (using flatMap()). I wanted to demonstrate how easy it is to combine multiple REST calls into one with RxJava + Retrofit.

**Old, Slow Code**

It's neat that Retrofit can return Observables, but what if you've got another library that doesn't support it? Or some internal code you want to convert to Observables? Basically, how do you connect old code to new without rewriting everything?

Observable.just() and Observable.from() will suffice for creating an Observablefrom older code most of the time:

private Object oldMethod() { ... }

public Observable<Object> newMethod() {

return Observable.just(oldMethod());

}

That works well if oldMethod() is fast, but what if it's slow? It'll block the thread because you're calling oldMethod() before passing it to Observable.just().

To get around that problem, here's a trick I use all the time - wrapping the slower part with defer():

private Object slowBlockingMethod() { ... }

public Observable<Object> newMethod() {

return Observable.defer(() -> Observable.just(slowBlockingMethod()));

}

Now, the Observable returned won't call slowBlockingMethod() until you subscribe to it.

**Lifecycle**

I saved the hardest for last. How do you handle the Activity lifecycle? There are two issues that crop up over and over again:

1. Continuing a Subscription during a configuration change (e.g. rotation).

Suppose you make REST call with Retrofit and then want to display the outcome in a ListView. What if the user rotates the screen? You want to continue the same request, but how?

1. Memory leaks caused by Observables which retain a copy of the Context.

This problem is caused by creating a subscription that retains the Contextsomehow, which is not difficult when you're interacting with Views! If Observable doesn't complete on time, you may end up retaining a lot of extra memory.

Unfortunately, there are no silver bullets for either problem, but there are some guidelines you can follow to make your life easier.

The first problem can be solved with some of RxJava's built-in caching mechanisms, so that you can unsubscribe/resubscribe to the same Observablewithout it duplicating its work. In particular, cache() (or replay()) will continue the underlying request (even if you unsubscribe). That means you can resume with a new subscription after Activity recreation:

Observable<Photo> request = service.getUserPhoto(id).cache();

Subscription sub = request.subscribe(photo -> handleUserPhoto(photo));

// ...When the Activity is being recreated...

sub.unsubscribe();

// ...Once the Activity is recreated...

request.subscribe(photo -> handleUserPhoto(photo));

Note that we're using the *same* cached request in both cases; that way the underlying call only happens once. Where you store request I leave up to you, but like all lifecycle solutions, it must be stored somewhere outside the lifecycle (a retained fragment, a singleton, etc).

The second problem can be solved by properly unsubscribing from your subscriptions in accordance with the lifecycle. It's a common pattern to use a CompositeSubscription to hold all of your Subscriptions, and then unsubscribe all at once in onDestroy() or onDestroyView():

private CompositeSubscription mCompositeSubscription

= new CompositeSubscription();

private void doSomething() {

mCompositeSubscription.add(

AndroidObservable.bindActivity(this, Observable.just("Hello, World!"))

.subscribe(s -> System.out.println(s)));

}

@Override

protected void onDestroy() {

super.onDestroy();

mCompositeSubscription.unsubscribe();

}

For bonus points you can create a root Activity/Fragment that comes with a CompositeSubscription that you can add to and is later automatically unsubscribed.

A warning! Once you call CompositeSubscription.unsubscribe() the object is unusable, as it will automatically unsubscribe anything you add to it afterwards! You must create a new CompositeSubscription as a replacement if you plan on re-using this pattern later.

Solutions to both problems involve adding code; I'm hoping that someday a genius comes by and figures out how to solve these problems without all the boilerplate.

**Conclusion?**

There isn't one yet for Android. RxJava is still rather new and adoption of it on Android is even newer. People are still figuring this stuff out; RxAndroid is in active development and there aren't any great samples out there yet. I bet that a year from now some of the advice I've given here will be considered quaint.

In the meantime, I find that RxJava not only makes coding easier but a bit more fun. If you're still not convinced, find me sometime and we'll talk about it over a beer.

*Thanks again to*[*Matthias Kay*](http://mttkay.github.io/)*for proofreading this article. Join him in making [RxAndroid](https://github.com/ReactiveX/RxAndroid) awesome!*