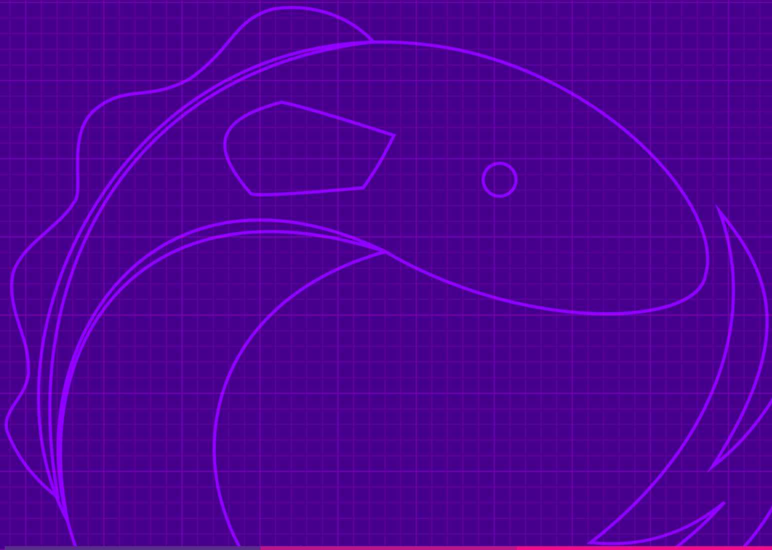
Reactive Programming on Android with RxJava



MYNAH SOFTWARE

Chris Arriola
Angus Huang

Reactive Programming on Android with RxJava

Christopher Arriola and Angus Huang

This book is for sale at http://leanpub.com/reactiveandroid

This version was published on 2017-06-27

This book is dedicated to my wife, Bianca. You inspire me beyond measure. - Chris Arriola

Contents

Chapter 1: What is Reactive Programming? **1**
The History of Reactive Programming 2
An Example 4

Chapter 2: RxJava Core Components **11**
The 3 O’s: Observable, Observer, and Operator 13
Marble Diagrams 17
Creating an Observable 19
Cold vs. Hot Observable 21
Lazy Emissions 22

Chapter 1: What is Reactive Programming?

Reactive programming can most simply be defined as *asynchronous programming with observable streams*. Well, what does *that* mean? Let's break it down…

Asynchronous programming, in the context of reactive programming, is a bit of a loaded term. In the traditional sense, it is programming in a non-blocking way such that long-running tasks are performed separately from the main application thread. In another sense, it is an event-driven style of programming where the events themselves are asynchronous and can arrive at any point in time.

Observable refers to an entity that can be subscribed to by any number of observers interested in its state. The observable entity will then push updates to its observers when there is a state change or event arrival. This is the classic Observer Pattern.

A *stream* (or *data stream*) can be thought of as an ordered sequence of events. These events may arrive at any point in time, may have no defined beginning or end, and are often generated by sources external to our application.

Re-assembling these terms, an *observable stream* is, then, a sequence of events that can be subscribed to and whose observers will be notified for each incoming event. And *asynchronous programming with observable streams* is a way to asynchronously handle data streams by using a push-based, observer pattern to keep the application responsive.

Can you think of some examples of data streams that we might want to handle using reactive programming? As an Android developer, you have no doubt dealt with many forms of data streams…

- **User-Generated Events**: Click events, swipe events, keyboard input, device rotations… these are just a few examples of events initiated by the user. If we consider each of these events across a timeline, we can see how they would form an ordered sequence of events–a dynamic and potentially infinite data stream.
- **I/O Operations**: Network requests, file reads and writes, database accesses… I/O operations are probably the most common type of data streams you will apply reactive principles to in practice. I/O operations are asynchronous since they take some significant and uncertain amount of time to complete. The response–whether it be JSON, a byte array, or some plain-old Java objects–can then be treated as a data stream.
- **External System Events**: Push notifications from the server, GCM broadcasts, updates from device sensors… events generated from external producers are similar to user-generated events in their dynamic and potentially infinite nature.

- **Just About Anything**: Really. Just about anything can be modeled as a data stream–that's the reactive mantra. A single, scalar value or a list of objects… static data or dynamic events… any of these can be made into a data stream in the reactive world.

All of the previous examples are events that we want our program to take immediate action on. That is the point of reactive programming–to be reactive. With reactive programming, data is a first-class citizen. Your program will be driven by the flow of data as opposed to the thread of execution. And RxJava, the library we will be using, provides a thorough set of APIs for dealing with these data streams in a concise and elegant manner. But before we delve into the anatomy of RxJava, let's take a look at how it came to be.

The History of Reactive Programming

On October 28, 2005, Microsoft CTO, Ray Ozzie, wrote a 5000-word internal memo titled "The Internet Services Disruption". The memo stressed to every department at Microsoft that they needed to adapt to the new Internet services era. With big players in the field like Google, Facebook, and Amazon, they needed to move fast.

Erik Meijer and the Cloud Programmability Team at Microsoft heeded that call. They were determined to alleviate the complexity that plagued these large systems and killed developer productivity. The team decided to work on a programming model that could be utilized for these data-intensive Internet services. The breakthrough occurred when they realized that by dualizing the Iterable interface from the Gang of Four's Iterator Pattern[1], they would have a nice push-based model for easily dealing with asynchronous data streams. Over the course of the next couple years, they would refine this idea and create Rx.NET (Reactive Extensions for .NET).

Rx.NET defined the set of interfaces (i.e. `IObservable`, `IObserver`) that would be fundamental to reactive programming. Along with these came a toolbox of APIs for manipulating data streams such as mapping, filtering, selecting, transforming, and combining. Data streams had achieved first-class status with Rx.NET.

The Birth of RxJava

One of the first users of this technology was Jafar Husain. When he left Microsoft and joined Netflix in 2011, he continued to evangelize Rx. Around that time, Netflix had successfully transitioned from a DVD rental service to an on-demand, streaming service. By 2012, business was booming–they would reach nearly 30 million subscribers by the end of the year. During this upward trajectory, the Netflix team realized that their servers were having a hard time keeping up with the traffic.

The problem was not simply in the sheer number of streaming subscribers but also in the myriad devices that Netflix supported. There were set-top boxes, smart TVs, game consoles, desktop

[1]https://en.wikipedia.org/wiki/Iterator_pattern

computers, and mobile devices. Each device had their own distinct UI/UX dictated by things like screen size, network bandwidth, input controls, and platform requirements. Netflix, at the time, had a generic, one-size-fits-all API for these clients to query. This meant that clients would often have to issue multiple queries to get all the required data and then piece them together to fit their particular UI. Not only would this be complex and slow on the client side, but it resulted in a lot of extra load on the server. Ben Christensen and his team decided to overhaul the Netflix API server architecture in order to decrease chattiness, increase performance, and allow for scalability.

They wanted their server API to be a "platform" for APIs and then hand over the reigns of implementing those APIs to the client teams. By having each client team develop their own custom endpoints, the clients could consolidate multiple calls into a single call and get the exact data they needed. This eliminated not only the latency of multiple network calls on the client but also load on the server.

The client teams were not expert server developers, however, and writing server APIs is not a trivial exercise. Performant servers usually need to utilize multiple threads to service simultaneous requests and issue concurrent requests to backend services and databases. The problem is… concurrent programming is difficult. The server team needed to make the process as simple and error-proof as possible for the client teams.

Luckily, Jafar Husain was there to preach the merits of Rx to Ben Christensen, who realized it was a great approach to addressing all of the problems mentioned above. Among other things, reactive programming abstracts threading away so that developers do *not* have to be experts in writing concurrent programs.

Because the Netflix APIs were implemented in Java, they began to port the Reactive Extensions to Java. They called it RxJava[2] and–in Netflix-fashion–open-sourced it. They stayed as close to the Rx.NET implementation as possible while adjusting naming conventions and idioms to suit the Java programming language. In February 2013, Ben and Jafar announced RxJava to the world in a Netflix Tech Blog post[3].

Reactive on Android

Even though RxJava started with the Netflix servers, it found its way across the tech stack and across the industry. From backends to frontends, web services to mobile clients… anything event-driven was a good candidate to get a reactive makeover.

While making its way across the stack, Rx also made its way across programming languages. Ports to other languages were worked on as open-source projects under an umbrella project called ReactiveX[4]. Reactive extensions were implemented for just about all the popular programming languages you could imagine: C++, Scala, Ruby, Python, Go, Groovy, Kotlin (go check out our other

[2]https://github.com/ReactiveX/RxJava
[3]https://medium.com/netflix-techblog/reactive-programming-in-the-netflix-api-with-rxjava-7811c3a1496a
[4]http://reactivex.io/

book[5]!), PHP, and Swift. There are even extensions for various frameworks, such as RxCocoa and RxAndroid (the latter of which we will no doubt discuss in this book).

The event-driven nature of Android made it an ideal platform for reactive programming. Android developers constantly have to deal with dynamic events in a responsive manner while making sure not to block the UI thread. Existing constructs like `AsyncTask` are both verbose and inadequate. And so RxJava's popularity in the Android world is no fluke.

Reactive Manifesto

Another contributor to the bump in popularity for reactive programming was the Reactive Manifesto[a]. This was a document written by Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson in 2013 (and updated in 2014). The manifesto argued that our present day's demand for increasing amounts of data and faster response times required a new software architecture–a reactive one. They defined a *Reactive System* to be one that is Responsive, Resilient, Elastic, and Message-Driven.

To date, nearly 20,000 people have signed the document. However, we leave the Reactive Manifesto here as an aside in this book since it only looks at reactive programming from a 10,000-foot view rather than really discussing how to accomplish it.

[a] http://www.reactivemanifesto.org/

An Example

Let's take a look at some RxJava in action:

```
Observable<Car> carsObservable = getBestSellingCarsObservable();

carsObservable.subscribeOn(Schedulers.io())
    .filter(car -> car.type == Car.Type.ALL_ELECTRIC)
    .filter(car -> car.price < 90000)
    .map(car -> car.year + " " + car.make + " " + car.model)
    .distinct()
    .take(5)
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(this::updateUi);
```

The above code will get the top 5 sub-$90,000 electric cars and then update the UI with that data. There's a lot going on here, so let's see how it does this line-by-line:

[5] https://leanpub.com/reactiveandroidrxkotlin

- Line 1: It all starts with our `Observable`, the source of our data. The `Observable` will wait for an observer to subscribe to it, at which point it will do some work and push data to its observer. We intentionally abstract away the creation of the `carsObservable` here, but we will cover how to create an `Observable` in the next chapter. For now, let's assume the work that the `Observable` will be doing is querying a REST API to get an ordered list of the top-selling cars.

Lines 3 - 10 form essentially a data pipeline. Items emitted by the `Observable` will travel through the pipeline from top to bottom. Each of the functions in the pipeline are what we call an `Operator`; `Operators` modify the `Observable` stream, allowing us to massage the data until we get what we want. In our example, we start with an ordered list of all top selling car models. By the end of our pipeline, we have just the top 5 selling electric cars that are under $90,000. The massaging of the data happens as follows:

- Line 3: `.subscribeOn(...)` tells the `Observable` to do its work on a background thread. We do this so that we don't block Android's main UI thread with the network request performed by the `Observable`.
- Line 4: `.filter(...)` will filter the stream down to only items that represent electric cars. Any `Car` object that does not meet this criteria will be discarded at this point and will not continue down the stream.
- Line 5: `.filter(...)` will further filter the electric cars to those below $90,000.
- Line 6: `.map(...)` will transform the element from a `Car` object to a `String` consisting of the year/model/make. From here on, this `String` will take the place of the `Car` in the stream.
- Line 7: `.distinct()` will remove any elements that we've already encountered before. Note, that this uniqueness requirement applies to our `String` values and not our `Car` instances, which no longer exist at this point in our chain because of the previous `.map(...)` call.
- Line 8: `.take(5)` will ensure that at most 5 elements will be passed on; if 5 elements are emitted, the stream will complete and emit no more items.
- Line 9: `.observeOn(...)` switches our thread of execution back to the main UI thread. So far, we've been working on the background thread specified in Line 3. Now that we need to manipulate our Views, however, we need to be back on the UI thread.
- Line 10 `.subscribe(...)` is both the beginning and end of our data stream. It is the beginning because `.subscribe()` prompts the `Observer` to do its work and start emitting items. However, the parameter we pass to it is the `Observer`, which represents the end of the pipeline and defines some action to perform when it receives an item (in our case, the action will be to update the UI). The item received will be the result of all of the transformations performed by the upstream `Operators`.

RxJava's Essential Characteristics

Taking our example, let's take a step back and examine the defining characteristics of RxJava.

Observer Pattern

RxJava is a classic example of the Observer Pattern. We start with the `Observable`, which is the source of our data. Then we have one or more `Observers`, which subscribe to the `Observable` and get notified when there is a new event. This allows for a push-based mechanism, which is usually far more ideal than continually polling for new events. RxJava adds to the traditional Observer Pattern, however, by also having the `Observable` signal *completion* and *errors* along with regular events, which we will see in Chapter 2.

Iterator Pattern

With a traditional Iterator Pattern, the iterator *pulls* data from the underlying collection, which would implement some sort of Iterable interface. What Erik Meijer essentially did in creating Rx was flip the Iterator Pattern on its head, turning it into a *push*-based pattern.

The `Observable` was designed to be the dual of the `Iterable`. Instead of pulling data out of an `Iterable` with `.next()`, the `Observable` pushes data to an `Observer` using `.onNext()`. So, where the Iterator Pattern uses synchronous pulling of data, RxJava allows for asynchronous pushing of data, allowing code to be truly reactive.

Functional Programming

One of the most important aspects of RxJava is that it uses functional programming, in particular, with its `Operators`. *Functional programming* is programming with pure functions. A *pure function* is one that satisfies the following two conditions:

1. The function always returns the same value given the same input (i.e. it's *deterministic*). This implies that the function cannot depend on any global state (e.g. a Java class's member variable) nor any external resource (e.g. a web service).
2. The function does not cause any side effects. This means that the function cannot mutate any input parameter, update any global state, or interact with any external resource.

Functional programming leads to code that is…

- **Declarative**: Instead of dictating *how* something is done, as is done with imperative programming, RxJava uses a declarative approach by describing *what* should be done (through our use of `Operators`). This declarative style maps much more closely to how we as humans think (as opposed to how a computer executes), and allows us to much more easily reason about what the program is doing.
- **Thread-Safe**: Because functional programming avoids state mutation, it is inherently thread-safe. We have none of the problems that we normally see when introducing concurrency (e.g. synchronization issues, race conditions, resource contention). Concurrency can be supported safely and easily with RxJava.

- **Testable**: Functional code becomes much easier to test because the functions are completely self-contained and deterministic. We don't need to mock the global state before executing unit tests; all we need is a set of input and expected output.

Note that not all RxJava `Operators` are totally *pure*. Case in point are the "do" methods (e.g. `.doOnNext()`), which were designed with the express purpose of letting the developer inject side effects, such as logging to the console, in order to examine the data stream at any point in the chain.

Note that no program can be functional forever. At some point, it needs to be imperative and spell out to the computer exactly *how* to do something, and it needs to modify state (whether that is updating a View, writing to the server, or just printing to console). RxJava just provides us a nice abstraction over the imperative, allowing us to program functionally. The idea is to implement as much of the program as you can using a functional approach and push out the imperative stuff to the very furthest edges of the program where being imperative becomes unavoidable. With the bulk of our program being functional, it will be more declarative, thread-safe, testable, and hopefully, free of bugs.

Conciseness

For how much it's doing, the code above is extremely concise. Imagine the number of lines that would be required to write this without RxJava. Indeed, a huge reason why the code is so concise is the use of lambdas. RxJava's functional style and lambdas go hand in hand. We recommend you use lambdas in your Java/Android project regardless, but *especially* so if you are using RxJava. If you are not yet using Android Studio 3.+, we recommend using the Retrolambda[6] library to add lambda support to your project. RxJava really isn't all that it can be without lambdas.

The second reason for its conciseness is the library of `Operators` that RxJava provides. These `Operators` provide extremely helpful functions that would otherwise take many lines of code to implement.

Thirdly, RxJava `Operators` are designed to be chained together, which also contributes to the conciseness of the resulting code. Each `Operator` method itself returns a modified `Observable` allowing for a subsequent `Operator` to be chained.

[6]https://github.com/evant/gradle-retrolambda

Fluent Interface

Although the above code snippet probably requires an initial somewhat-lengthy explanation, once you have a baseline understanding, you can see the code is very idiomatic and self-explanatory. RxJava uses a fluent interface with `Operator` names that read like English prose, especially when chained together. This allows for not only faster coding, but also for someone reading the code to parse what's going on much more quickly.

Lazy Evaluation

RxJava, in general, uses lazy evaluation. Remember from above that the network call is not performed until the very last step when we make the `.subscribe()` call. The `Observable` lays idle until an `Observer` subscribes to it. Because subscription initiates the action, the `Observable` can be reused; every `Observer` that subscribes will cause the `Observable` to be invoked.

Lazy evaluation has its pros and cons. In many cases, lazy evaluation is beneficial. For example, if the `Observable` queries a web service, we likely want the most up-to-date data, which means that execution should happen when the `Observer` subscribes (and not when the `Observable` is created). The same should apply for any `Observer` that subscribes to the `Observable`; no matter what time the `Observer` subscribes, it should get the latest data.

However, if our `Observable` is retrieving data that will not change, then we do not want it to make a network round-trip for every subscriber. We'd prefer eager evaluation in this case and have the `Observable` get the data once and hold on to it. RxJava, and its plentiful toolbox of APIs, actually allows for this with the `.cache()` operator. In fact, we'll see how to dictate the laziness vs. eagerness of an `Observable` more when we talk about `Observable` creation and cold vs. hot `Observables` in Chapter 2.

Easy Concurrency

In our example, we were able to have the network request executed on a background thread and then switch back to having the UI be updated on the main UI thread. This was accomplished with literally two lines of code (using our `.subscribeOn()` and `.observeOn()` `Operators`). We didn't have to manually create a new Thread; we didn't have to implement an AsyncTask class and all its methods; we didn't have to lock or synchronize anything. This is concurrency made easy. We will take a deeper dive into concurrency in Chapter 4: Multithreading.

Async Error Handling

Java's regular error-handling mechanism of throwing/catching exceptions is ill-equipped at handling concurrency. What happens when the work is performed on a separate thread? Where would the exception be propagated to? The `Observer` does not have a chance to catch and handle this exception (unless it occurs before the `.subscribe()` returns, which is unlikely in a non-blocking function where all the interesting work is performed on a separate thread *after* the function returns).

RxJava provides a solution by letting the `Observer` provide an error callback. In RxJava, errors are passed down the stream just like any other event. This allows the `Observer` to be reactive in the face of failure in the same way it is reactive with normal events. Also, it does away with the verbose try/catch construct, which most Java developers would love not having to deal with. We will discuss more about error handling in Chapter 7.

Functional Reactive Programming (FRP)

There has been a lot of confusion over terminology around the industry. While reactive programming includes elements of functional programming, it is debatable whether we can call it "functional reactive programming". *Functional Reactive Programming (FRP)* is a precise, mathematically defined programming technique defined in 1997 by Conal Elliott and Paul Hudak. And Rx does not conform to FRP's precise definition or "continuous time" requirement where values change continuously over time.

Still, a lot of different resources will still refer to Rx as "functional reactive programming", much to the discontent of Conal Elliott. The FRP creator has been everywhere from StackOverflow to Twitter to call out misappropriations of his term, but "misuse" of the term has only spread along with Rx's popularity. Most of these Rx articles do so without knowledge of the original FRP specification, although some do so in outright disregard of it because it is such a convenient term to use when describing Rx.

This book will refrain from describing Rx as "functional reactive programming" or "FRP". But we did want to clarify the backstory and let you make your own decision on how to use the term and how to interpret it as you come across it in other literature.

As you proceed through this book, keep in mind that there is a significant learning curve to reactive programming. It requires a different way of thinking as compared to the regular imperative way of doing things that most Java programmers are used to. But as with most changes in thinking, hopefully there's that a-ha moment where things start to click.

RxJava was designed to handle asynchronicity and event-driven conditions in a concise, functional, resilient, and reactive way. The more complex your data streams are and the more inter-dependent they are, the more value you'll get out of using RxJava. When data streams (such as network responses) need to be combined or nested, the old imperative way gets messy quickly; you'll quickly find yourself in "callback hell" trying to manage concurrency-related complexities. RxJava is the way out of this hell.

If you're lucky enough, however, to have a simple program with completely independent data streams, then you may find RxJava to be overkill. Being direct and going imperative might be the way to go in this case rather than dealing with the RxJava abstraction layer.

Regardless, it is worthwhile to learn reactive programming. RxJava is now pervasive across platforms–from backend servers to mobile clients–and it has definitely cemented its place in the

Android ecosystem. RxJava is supported in many open-source Android libraries (Retrofit[7], to name one very important one). It's being embraced by some of the top Android apps in the Play Store. And Google even designed the new Android Architecture Components[8] with RxJava in mind, embracing many of its push-based, reactive concepts (the `LiveData` class was no doubt based on RxJava's `Observable`–there is an API to convert between the two–and the Room persistence library has an option to return an RxJava `Flowable`). And as an open-source library itself, RxJava is constantly being improved and adapted to developers' needs.

As you can see, it will be difficult to avoid RxJava as an Android developer. It's time to embrace it. Learning RxJava will arm you with a powerful tool whether you choose to use it or not. Most likely, it will come in quite handy with all the complexities that arise in the real world. So let's dive right in.

[7]http://square.github.io/retrofit/

[8]https://developer.android.com/topic/libraries/architecture/index.html

Chapter 2: RxJava Core Components

In the RxJava world everything can be modeled as a stream–a stream emits item(s) over time, and each item can be transformed or modified as it passes through. An observer or consumer can then subscribe and perform an action from each emission returned by the stream. Further, streams in RxJava are highly composable and can even be combined with other streams to produce a desired result.

If you think about it, a stream is not a new concept. A `Collection` in Java can be modeled as a stream where each element in the `Collection` is an item emitted in the stream. On Android, click events can be a stream, location updates can be a stream, push notifications can be a stream, and so on.

Traditionally, processing data streams in Java is done imperatively. For example, given a list of `User` objects, say we want to return only those that have a blog. That function might look something like:

```java
/**
 * Returns all users with a blog.
 * @param users the users
 * @return users that have a blog
 */
public static List<User> getUsersWithABlog(List<User> users) {
    List<User> filteredUsers = new ArrayList<>();
    for (User user : users) {
        if (user.hasBlog()) {
            filteredUsers.add(user);
        }
    }
    return filteredUsers;
}
```

The above code might look very familiar to you: a loop that iterates through each item in the provided collection; an if-statement to check if a condition is true; and for those that pass, the necessary action is performed (i.e. the item is added to the returned list).

Processing data streams is so commonplace that in Java 8 the package **java.util.stream** was introduced. Essentially, the goal of Java 8 Streams is to increase the level of abstraction when dealing with streams by providing a declarative way of doing so over the traditional imperative way. In the imperative way shown above, we specified *how* to process a stream; but with a declarative approach, we would simply specify *what* we want to do to that stream. This way of looking at stream processing is ubiquitous in functional programming languages and is now introduced, along with a few other functional programming constructs, as part of the core language. Using streams, our above code would now look like:

```java
/**
 * Returns all users with a blog.
 * @param users the users
 * @return users that have a blog
 */
public static List<User> getUsersWithABlog(List<User> users) {
    return users.stream()
                .filter(user -> user.hasBlog())
                .collect(Collectors.toList());
}
```

Using streams, the same operation can be done much more concisely. First, we convert the `List` into a `Stream`, filter the `Stream` by users that have a blog, and finally convert the `Stream` back into a `List`.

Now you might be wondering, if everything is a stream in RxJava and Java 8 supports streams, why not just use Java 8's streams? Although there are many reasons to prefer RxJava as we'll see in the book, at a very high-level, unlike Java 8 streams, RxJava enables us to do true reactive programming. The toolbox of functional language constructs such as *map*, *filter*, and *merge* might be shared by both, however, RxJava's version of a stream is much easier to work with, and on top of that, shines when dealing with asynchronicity.

Using our example, say the call `user.hasBlog()` is a network operation that blocks the current thread until a response is received (probably not the desired implementation in production, but for educational purposes, say this were true). All things equal, both the imperative and Java 8 stream approach of retrieval would block the thread invoking the method `.getUsersWithABlog(List<User>)`. Thus, the method cannot be called from the UI thread and some sort of approach that calls this method from a background thread is required.

Using RxJava, the solution to this problem is trivial.

```java
/**
 * Returns all users with a blog.
 * @param users the users
 * @return an Observable emitting users that have a blog
 */
public static Observable<User> getUsersWithABlog(List<User> users) {
    return Observable.fromIterable(users)
            .filter(user -> user.hasBlog())
            .subscribeOn(Schedulers.io());
}
```

There are a few new classes here that we will definitely dive into, but first notice how the readability has not been dramatically affected despite the operation now running on a background thread (i.e. through `.subscribeOn(Schedulers.io())`. In essence, we have declaratively specified the thread in which the operation should occur on. We did, however, modify the signature of the function by returning an `Observable<User>`. As we will see, this is the first step towards reactive programming on Android.

The 3 O's: Observable, Observer, and Operator

The concept of a stream is modeled in RxJava using 3 main constructs which I like to call the "3 O's". These are the `Observable`, `Observer` and the `Operator`. The `Observable` is an entity that emits item(s) over time (i.e. the stream). It can then be *subscribed* to at any point in time by an `Observer` which will receive items that were pushed down along the stream. An `Operator`, or sequence of `Operators`, can then be inserted in between the `Observable` and `Observer` to perform actions that transform, filter, aggregate, and combine data emitted down the stream.

> The 3 O's of RxJava are the `Observable`, `Observer` and `Operator`. An `Observable` emits items over time, an `Observer` subscribes to receive those items, and `Operators` can be used to transform items emitted by the `Observable` including the `Observable` itself.

Observable and Observer Pair

An `Observable` can emit any number of items. As mentioned, to receive or listen to these emitted items, an `Observer` needs to *subscribe* to the `Observable`. The `.subscribe()` method is defined by the `ObservableSource` interface, which is implemented by `Observable`.

```java
public interface ObservableSource<T> {
    void subscribe(Observer<? super T> observer);
}
```

Once the `Observable` and `Observer` have been paired via `.subscribe()`, the `Observer` will receive the following events through its defined methods:

- **.onSubscribe()**: this method is called along with a `Disposable` object which may be used to *unsubscribe* from the `Observable` to stop receiving items
- **.onNext()**: this method is called when an item is emitted by the `Observable`
- **.onComplete()**: this method is called when the `Observable` has finished sending items
- **.onError()**: this method is called when an error is encountered within the `Observable`; the specific error is passed to this method

```java
public interface Observer<T> {
    void onSubscribe(Disposable d);

    void onNext(T value);

    void onError(Throwable e);

    void onComplete();
}
```

`.onNext()` can be invoked 0, 1, or multiple times by the `Observable` whereas `.onComplete()` and `.onError()` are considered terminal events, meaning, if either of the two methods were called, no further method would be invoked on the `Observer`. This is a crucial contract of an `Observable` that must be enforced when creating an `Observable`.

The method `.subscribe()` is also an overloaded method of an `Observable`. Depending on the `Observable` being subscribed to, sometimes it is not necessary to provide a "full" `Observer` when subscribing. For example, if we know that the `Observable` will never invoke `.onComplete()` in the case of an infinite stream, we can choose one of the other overloaded methods. The other options are as follows (note that all these versions will return a `Disposable` object):

- **.subscribe()**: all methods are ignored. All errors will forward to the `RxJavaPlugins.onError()` handler (more information on this in Chapter 7: Error Handling).
- **.subscribe(Consumer<? super T> onNext)**: only `.onNext()` events are received. All errors will forward to the `RxJavaPlugins.onError()` handler.
- **.subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError)**: only `.onNext()` and `.onError()` events are received and `.onComplete()` is ignored.
- **.subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError, Action onComplete)**: all events are received.
- **.subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError, Action onComplete, Consumer<? super Disposable> onSubscribe)**: all events are received; additionally, a `Consumer` can be specified to receive the upstream's `Disposable` object.

Previously in RxJava 1.x, if a method was omitted but called by the `Observable` (e.g. the `.onComplete()` definition was not provided on subscription but called by the `Observable`), an exception would have been thrown. However, this was changed in RxJava 2 so that you don't have to provide a method definition if you are not concerned with certain events. Although the API is more forgiving now, you should still use an overloaded `.subscribe()` method with caution. In most cases, you would want to provide an `.onError()` definition and handle the error as appropriate for your use case.

Push vs. Pull

Looking at the `Observable` and `Observer` definitions, we can see that the design is inherently a *push* based system. An `Observer` will *react* upon receiving an item that was *pushed* to it by the `Observable` that it is subscribed to. However, this does not imply that emitted events are asynchronous. In fact, by default an `Observable` is synchronous–events will be emitted in the `Observable` stream on the same thread where `.subscribe()` is invoked.

```java
Log.d(TAG, "Creating Observable.");
Observable.create((ObservableOnSubscribe<String>) emitter -> {
    emitter.onNext("test");
    emitter.onComplete();
}).subscribe(new Observer<String>() {
    @Override
    public void onSubscribe(Disposable d) {
    }

    @Override
    public void onNext(String value) {
        Log.d(TAG, "onNext(): " + value);
    }

    @Override
    public void onError(Throwable e) {
    }

    @Override
    public void onComplete() {
        Log.d(TAG, "onComplete()");
    }
});
Log.d(TAG, "After subscribing.");
```

The above code snippet will display in the console:

```
Creating Observable.
onNext() - test
onComplete()
After subscribing.
```

The resulting order of print statements is observed since RxJava does not specifically impose asynchronous behavior. It is not opinionated about where the asychronicity originates unless it is otherwise specified.

> RxJava is synchronous by default. It is not opinionated about where asynchronicity originates; it must be explicitly specified.

As you might guess, synchronous behavior may not be desired, and in many real-world cases, we would want the underlying `Observable` to operate on a separate thread from the calling thread. The power of RxJava lies in dealing with asynchronous streams, and we will look at how to do multi-threading in RxJava in Chapter 4 - Multithreading. For now though, it is important to understand that just because an `Observer` receives items via *push* does not mean that any sort of concurrency is imposed.

RxJava also supports *pulling* from an `Observable`. Although this is not idiomatic RxJava, it was included primarily for interoperability with codebases that aren't 100% fully adapted to be reactive. We will look at this more in Chapter 5 - Reactive Modeling on Android.

Operator

`Operators` are very powerful constructs that allow us to declaratively modify item emissions of an `Observable` including the `Observable`/s themselves. Through `Operators`, we can focus on the business logic that make our applications interesting rather than concerning ourselves with low-level details of an imperative approach. Some of the most common operations found in functional programming (such as *map*, *filter*, *reduce*, etc.) can also be applied to an Observable stream. Let's take a look at `.map()` as an example:

```
Observable<Integer> intObservable =
    Observable.create((ObservableOnSubscribe<Integer>) emitter -> {
        emitter.onNext(1);
        emitter.onNext(2);
        emitter.onNext(3);
        emitter.onNext(4);
        emitter.onNext(5);
        emitter.onComplete();
    });

intObservable.map(val -> val * 3)
    .subscribe(i -> {
        // Will receive the following values in order: 3, 6, 9, 12, 15
    });
```

The code snippet above would take each emission from the `Observable` and multiply each by 3, producing the stream 3, 6, 9, 12, 15; but say we wanted to only receive even numbers. This can be achieved simply by chaining a `.filter()` operation.

```java
intObservable.map(val -> val * 3)
    .filter(val -> val % 2 == 0)
    .subscribe(i -> {
        // Will receive the following values in order: 6, 12
    });
```

As you can see from these examples, we were able to chain the operator `.filter()` and subsequently subscribe to the stream. This is because RxJava was intentionally designed to have a fluent interface[9]. A "fluent interface", as coined by Martin Fowler and Eric Evans, is a style of interface such that the return type of an object's method is the same type as the object, or another type depending on the action. In RxJava this means that applying an operator to an `Observable` will return an `Observable` or another base reactive type. In other words, with a fluent interface design we are able to perform operator method chaining which dramatically improves readability.

The astute reader might wonder: "what's the point in using an operator? Why not just apply the multiplication and check if the resulting number is even in the observer? Wouldn't that be simpler?"

```java
intObservable.subscribe(i -> {
    int multipliedVal = i * 3;
    if (multipliedVal % 2 == 0) {
        // Will receive the following values in order: 6, 12
    }
});
```

Indeed, the above code is functionally equivalent to applying the `.map()` and `.filter()` operators, however, this approach is not encouraged. Not only is it not idiomatic reactive programming (i.e. only values that you are actually interested in receiving should go through the `Observer`'s `.onNext()` method), as more transformations are required from the stream, the complexity of the code in the `Observer` significantly increases. But with the help of RxJava's rich set of `Operator`s, we are able to reason about data transformations in a much simpler way. We will look into some more examples in the next chapter, Chapter 3 - Operator Toolkit.

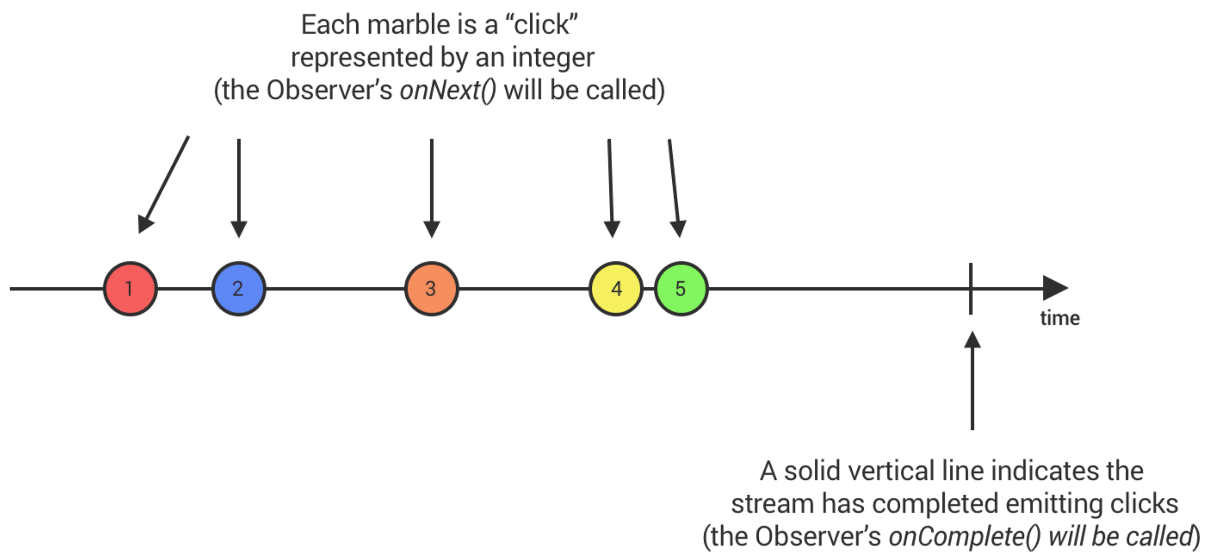
> A series of simple, single-purposed operators are encouraged in RxJava. Data that arrives in an `Observer`'s `.onNext()` should be in its final processed form.

Marble Diagrams

Marble diagrams are a common aid in visualizing `Observables`. Such diagrams are extensively used throughout the RxJava wiki[10] and we will see several of them throughout the book. Say we have an `Observable` emitting view clicks which are represented by an integer value:

[9] https://www.martinfowler.com/bliki/FluentInterface.html

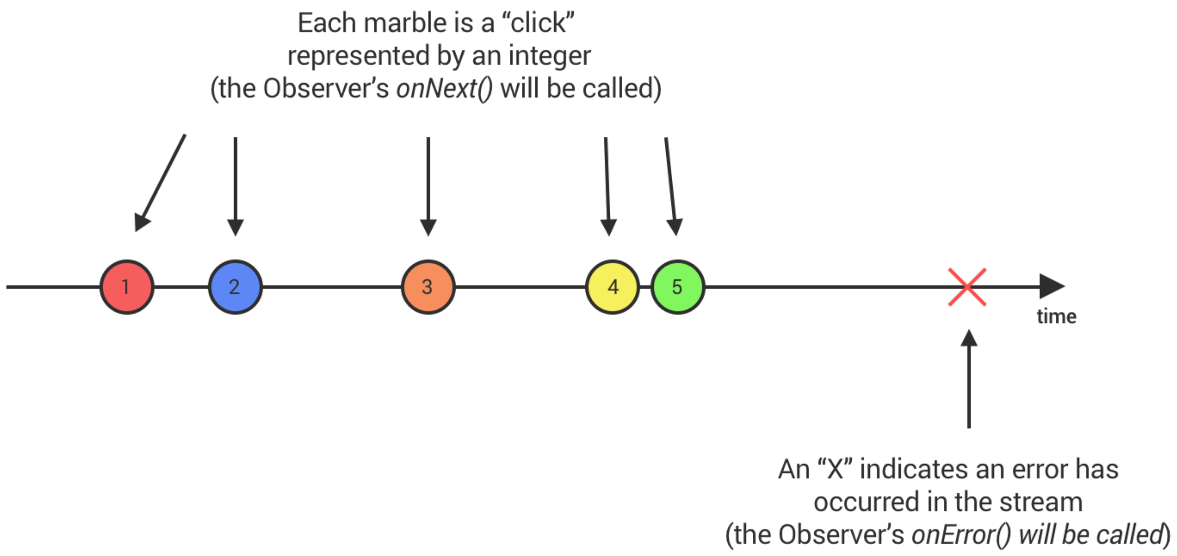
[10] https://github.com/ReactiveX/RxJava/wiki


Each marble is a "click"
represented by an integer
(the Observer's *onNext()* will be called)
1
2
3
4
5
time
A solid vertical line indicates the
stream has completed emitting clicks
(the Observer's *onComplete() will be called*)

Marble diagram of an Observable - Completed Successfully

The above marble diagram depicts clicks propagated down the stream as it is received over time. The clicks are represented by an integer (the value of which doesn't matter) and the stream completes by invoking an `Observer`'s `.onComplete()` method. In Android, this successful completion might mean that the user has backgrounded the app or completed the desired action on the screen, and thus click events are no longer received.

On the other hand, say an error occurred to the `Observable` while view click events were being propagated. For example, a view animation failed and so no further clicks should be propagated. In this case, an `Observer`'s `.onError()` method would be invoked and no further events should occur down the stream. In this event, the diagram would display an error using an "X".


Each marble is a "click" represented by an integer (the Observer's *onNext()* will be called)
1
2
3
4
5
time
An "X" indicates an error has occurred in the stream (the Observer's *onError() will be called*)

Marble diagram of an Observable - Error

Creating an Observable

Now that we've have a basic, high-level understanding of the different components of RxJava, let's dive into the `Observable` a bit more–specifically, different ways that we can create one. We've briefly glanced at the most verbose way of creating an `Observable` (i.e. `.create()`); however, there are simpler and more convenient APIs that are available to us. A few handy ones are:

Observable.just(T item)

`.just()` is one of the most common ways to wrap any Object to an `Observable`. It creates an `Observable` of type `T` that *just* emits the provided `item` via an `Observer`'s `.onNext()` and then completes via `.onComplete()`.`Observable.just()` is also overloaded and you can specify anywhere from 1 to 9 items to emit.

Example usage:

```
Observable.just(1, 2, 3).subscribe(val -> {
    // val will be 1, 2, 3
});
```

Observable.fromArray(T... items) and Observable.fromIterable(Iterable<? extends T> iterable)

`.fromArray()` and `.fromIterable()` are used to create an `Observable` from an array and an `Iterable` (e.g. `List`, `Map`, `Set`, etc.), respectively. Upon subscription, the resulting `Observable` will emit the items in the array or `Iterable` and complete after.

> Note that converting an array or `Iterable` can also be done using `.just()`; the generated `Observable` would then be of type `Observable<T[]>` or `Observable<List<T>>` instead of type `Observable<T>`. This is definitely allowed and sometimes necessary, however, it's a bit cumbersome since applying transformations require while/for loops through each item which undermines the power of RxJava. If possible, seek to have an `Observable` as "flat" as possible.

Observable.range(int start, int count) and Observable.rangeLong(long start, long count)

`.range()` and `.rangeLong()` will create an `Observable<Integer>` and `Observable<Long>`, respectively. Both will emit a range of values starting from `start` up to, but not including, `start + count`.

Observable.empty()

`.empty()` returns no items and immediately invokes an `Observer`'s `.onComplete()` method when subscribed to. By itself, it is not very useful and it's commonly used in conjunction with other operators.

Observable.error(Throwable exception)

`.error()` wraps an exception and invokes an `Observer`'s `.onError()` method when subscribed to.

Observable.never()

Creates an `Observable` that never invokes any of an `Observer`'s method when subscribed to. This is primarily used for testing purposes.

All of the above methods for creating an `Observable` can be replicated by using `.create()`. For example, `Observable.just()` can be implemented as:

```java
public static <T> Observable just(T item) {
    return Observable.create(emitter -> {
        emitter.onNext(item);
        emitter.onComplete();
    });
}
```

As you can see, `.create()` is much more powerful and flexible in terms of constructing an `Observable` and even allows you to violate the `Observable` contract (i.e. calling another event after a terminal event has been invoked). For this reason, it's much preferred to use any of the above `Observable` convenience creation methods as they are safer and easier to use.

Something to note about the aforementioned methods for creating `Observables` is that for each `Observer` that subscribes, each one will receive it's own independent stream from start to finish. In other words, all the emitted values from the subscribed `Observable` would be repeated. `Observables` that behave this way are called *cold* `Observables`.

Cold vs. Hot Observable

An `Observable` is considered cold if it is not actively emitting items; it only starts emitting items when it is subscribed to. Each subscription to a cold `Observable` will cause it to emit the underlying sequence from beginning to end for each `Observer`. Note, however, that the exact values of the sequence may differ for each `Observer`, depending on the underlying action the `Observable` encapsulates as we'll see in the next section.

Generally, a cold `Observable` is what you would want if you want a complete copy of events emitted in the stream. It is what you want for operations that should only be invoked when subscribed to: network operations, database queries, file I/O, etc.

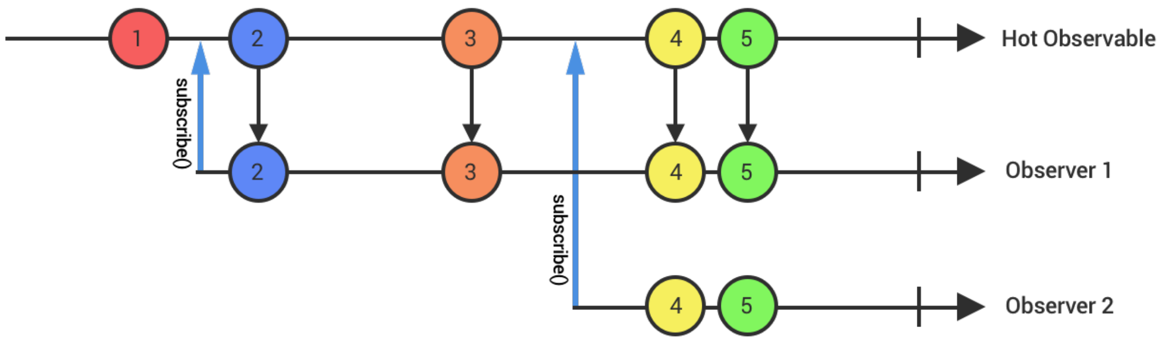
Looking at `.just()` as an example, we see that the sequence is repeated on each subscription:

```
Observable observable = Observable.just("A", "B", "C", "D", "E");

// observer 1
observable.subscribe(val -> {
    // "A", "B", "C", "D", "E" will be received in order
});

// observer 2
observable.subscribe(val -> {
    // Again, "A", "B", "C", "D", "E" will be received in order
});
```

While cold `Observables` are subscriber-dependent, hot `Observables` on the other hand, are not. They have their own timelines and actively emit items regardless if there are any subscribers. An `Observer` would receive events starting from the point of subscription and wouldn't receive any of the events emitted prior to subscription. Hot `Observables` model processes that are temporal: a click event, an event bus, etc.



Hot Observable

It is important to know whether or not a given `Observable` is cold or hot as it informs you on what type of operations you should consider. For example, you might want to avoid `Operators` that cache and aggregate emissions from hot `Observables` given that these type of `Observables` can in theory produce an unlimited number of events (see Flowable). We will take a look more at hot Observables in Chapter 5.

Lazy Emissions

An intrinsic property of a cold `Observable` is that it is *lazy*–the underlying sequence is only computed and emitted on subscription time. To demonstrate this, let's look at another `Observable` creation method `.fromCallable()`:

Observable.fromCallable(Callable<? extends T> callable)

`.fromCallable()` creates an `Observable` that when subscribed to, invokes the function supplied, and then emits the value returned by that function.

Constructing an `Observable` with `.fromCallable()` won't actually do the work specified inside the `Callable`. All it does is define the work to be done. As such, `.fromCallable()` is the ideal construction method to be used for any potentially long running UI-blocking operation. For example, we can wrap a network call returning a `User` object by returning the value of the network call inside the function provided to `.fromCallable()`.

```
Observable<User> observable = Observable.fromCallable(() -> {
    return apiService.getUserWithId(123);
});
observable.subscribe(user -> {
    // Got user with ID 123
});
```

Observable.defer(Callable<? extends ObservableSource<? extends T>> supplier)

`.defer()` is another `Observable` creation method available for deferring a potentially long running UI-blocking operation. The main distinction between `.defer()` and `.fromCallable()` is that the former returns an `Observable` in the supplied function.

`.defer()` creates an `Observable` that calls an `ObservableSource` factory to create an `Observable` for each new `Observer` that subscribes.

The above code for retrieving a user by ID can then be rewritten as:

```
Observable<User> observable = Observable.defer(() -> {
    return Observable.just(apiService.getUserWithId(123));
});
observable.subscribe(user -> {
    // Got user with ID 123
});
```

The main reason to use `.defer()` over `.fromCallable()` is if the creation of an `Observable` in itself is a blocking operation. However, these should in general be rare instances and `.fromCallable()` should suffice.

`.fromCallable()` and `.defer()` should be used if the evaluation of a value takes some time to compute and we would like to delay that computation up until the time of subscription. In contrast, if `.just()` is used we would not be delaying that computation up until the time of subscription, instead, the value would be computed immediately.

```
Observable<User> lazyObservable = Observable.fromCallable(() -> {
    return apiService.getUserWithId();
});

// At this point, network call has not yet been made.
lazyObservable.subscribe(user -> {
    // After subscribing, the network call is made and the user object is return\
ed
});

Observable<User> notLazyObservable = Observable.just(apiService.getUserWithId());

// At this point, the network call has already been made
// which blocks the calling thread.
notLazyObservable.subscribe(user -> {
    // After subscribing, the network call has already been
    // made on construction. The user object is still returned.
});
```

Although on subscription both `Observers` of **lazyObservable** and **notLazyObservable** receive the `User` object, the time at which the function was evaluated differs. The lazily evaluated approach is idiomatic RxJava and it also allows us to add asynchronous behavior whereas the other approach does not. We will look into how we can add asynchronous behavior in Chapter 4: Multithreading).

Lastly, one other thing to note about laziness is that although the action is repeated on each subscription, the actual emitted values of the sequence may differ. Using our example above, if the `User` object was modified on the server between subscriptions, one `Observer` would receive the older version of the `User` object, whereas the more recent `Observer` would receive the updated `User` object.