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*

Table of Contents

Introduction

Since its inception in 2012, RxJava has slowly gained in popularity for enabling reactive programming on Android. Today in 2017, it is now deemed as the go-to and leading reactive library. Many companies have adopted the reactive way of programming including Google with its release of Android Architecture Components, which has many reactive elements in its design.

In *Reactive Programming on Android with RxJava*, we seek to condense RxJava principles and provide a structured and simplified approach with a lot of code examples. This book seeks to serve as a foundation for experienced Android developers who are new to RxJava so that they can start integrating it into their apps.

The book is broken into two parts:

The 1st part of the book will go through the basics:

- In Chapter 1, we will give the backstory of RxJava and discuss what reactive programming is.
- In Chapter 2, we will examine the core components in RxJava.
- In Chapter 3, we will dive deeper into operators and highlight a couple of important ones.
- In Chapter 4, we will cover multithreading and concurrency.

The 2nd part of the book will go through advanced concepts:

- In Chapter 5, we will put together all the lessons from the 1st part of the book and see how they apply to Android.
- In Chapter 6, we will talk about backpressure to control the flow of data.
- In Chapter 7, we will go over error handling and how it differs from Java’s exception handling.

Feedback and Questions

For any feedback or questions, please reach out to us directly:

- Christopher Arriola (Email: c.rriola@gmail.com, Twitter: @arriolachris)
- Angus Huang (Email: angus.huang@alum.mit.edu)

Part 1: RxJava Basics

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, 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 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 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.

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. 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 book!), 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. 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.

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)
```

```
 5      .filter(car -> car.price < 90000)
 6      .map(car -> car.year + " " + car.make + " " + car.model)
 7      .distinct()
 8      .take(5)
 9      .observeOn(AndroidSchedulers.mainThread())
10      .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:

- 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`; `Operator`s 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 `Operator`s.

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 `Observer`s, 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 `Operator`s. *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 `Operator`s). 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 `Operator`s 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 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 `Operator`s that RxJava provides. These `Operator`s provide extremely helpful functions that would otherwise take many lines of code to implement.

Thirdly, RxJava `Operator`s 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.

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 `Observable`s 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()` `Operator`s). 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, 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 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.

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 `Operator`s, 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 `Operator`s 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`.

```
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

```
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

`Operator`s are very powerful constructs that allow us to declaratively modify item emissions of an `Observable` including the `Observable`/s themselves. Through `Operator`s, 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.

```
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. 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?"

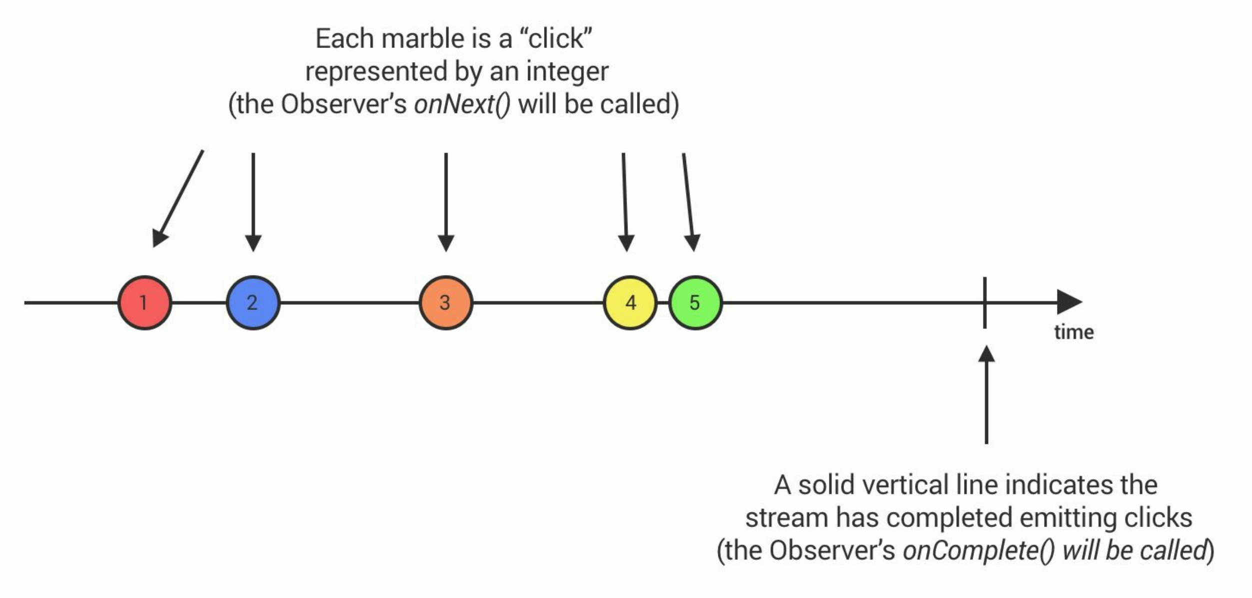
```
intObservable.subscribe(i -> {
    int multipliedVal = i * 3;
    if (multipliedVal % 2 == 0) {
        // Will receive the following values in order: 6, 12
    }
});
```

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

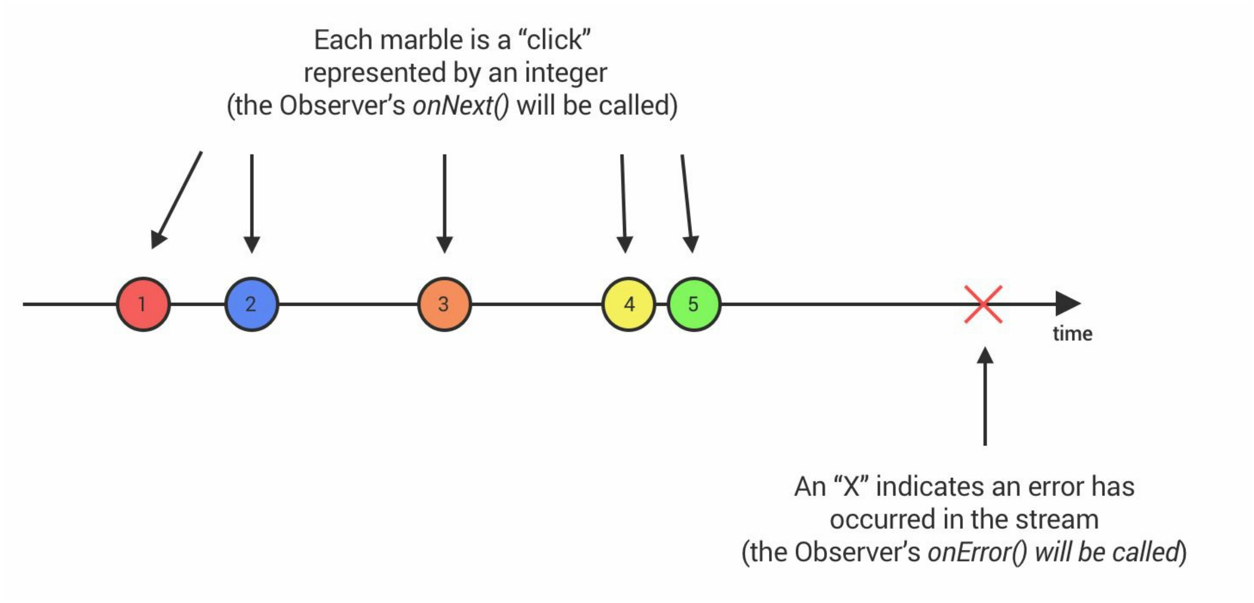
Marble diagrams are a common aid in visualizing `Observable`s. Such diagrams are extensively used throughout the RxJava wiki and we will see several of them throughout the book. Say we have an `Observable` emitting view clicks which are represented by an integer value:


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:

```
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 `Observable`s 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. `Observable`s that behave this way are called *cold* `Observable`s.

Cold vs. Hot Observable

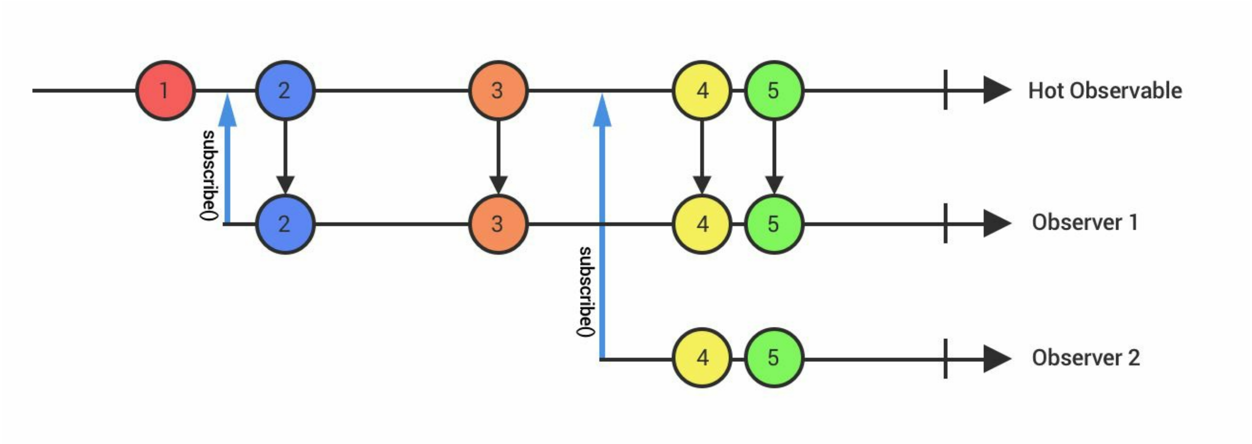
An `Observable` is considered cold if it is not actively emitting items; it only starts emitting items when it is subscribed to. Each subscription to a cold `Observable` will cause it to emit the underlying sequence from beginning to end for each `Observer`. Note, however, that the exact values of the 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:

```java
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 `Observable`s are subscriber-dependent, hot `Observable`s 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 `Observable`s 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 `Operator`s that cache and aggregate emissions from hot `Observable`s given that these type of `Observable`s 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 ret
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 `Observer`s 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.

Chapter 3: Operators

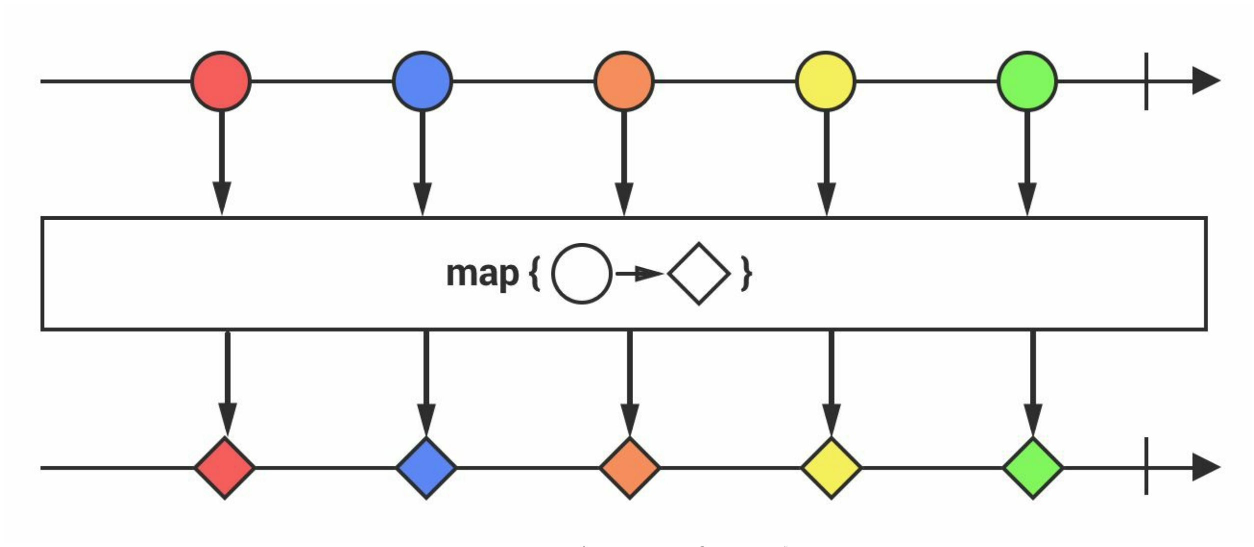
Now that we have established a foundation for the fundamentals of RxJava (i.e. the 3 O's), we can dive deeper into the rich set of `Operator`s that are available to us. Through the contract defined by the `Observable`, we are able to declaratively massage data emitted upstream to produce a desired result. Arguably, the set of `Operator`s and the ability to transform, combine, and aggregate items in such a way is one of the main reasons why RxJava is so powerful. In this chapter, we will look at some of the most commonly used `Operator`s and provide several examples along with marble diagrams to aid in visualizing how these `Operator`s work. Not all `Operator`s will be covered (as there are too many to cover); however, after reading this chapter, you should have a good foundation for how `Operator`s work and learning new ones should be a breeze.

Transforming and Filtering

`Operator`s that transform and/or filter are some of the handiest `Operator`s that are available as those do not require a timing or buffering aspect to it–the operations are applied as soon as a new element is produced in the stream. In the previous chapter, we looked at *map* and *filter* to apply a simple transformation and filtering of data. The following marble diagrams demonstrate how each one affects emitted items before they are propagated to subscribers.

Map

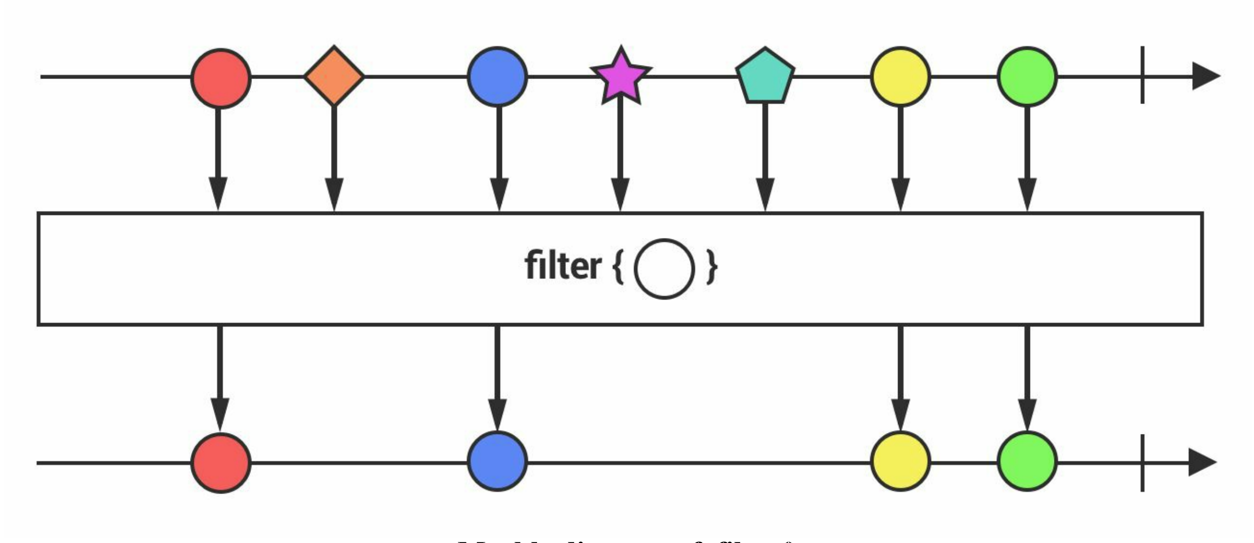
`.map()` works by applying the mapper function provided on each item emitted upstream and passes the return value from that function downstream.



Marble diagram of .map()

Filter

`.filter()` works by applying the predicate on each emission. If the return value of the predicate is *true*, the item will continue downstream, otherwise, it will be filtered out.



Marble diagram of .filter()

Say we had an `Observable` of `Post` objects that contain text and we are interested in processing a `Post`'s text only if it contains a certain amount of characters (e.g. below 160 characters).

```java
Observable<Post> postObservable = // ...
postObservable.map(post -> post.text)
    .filter(text -> text.length() < 160)
    .subscribe(text -> {
        // Receive text that is less than 160 characters
    });
```

Notice that the `.map()` and `.filter()` operators are kept short and succinct. Although the same action could have been done with a single `Operator` (i.e. just with `.filter()`), it is preferable to break up operations into smaller actions as it improves readability and keeps each operation simple.

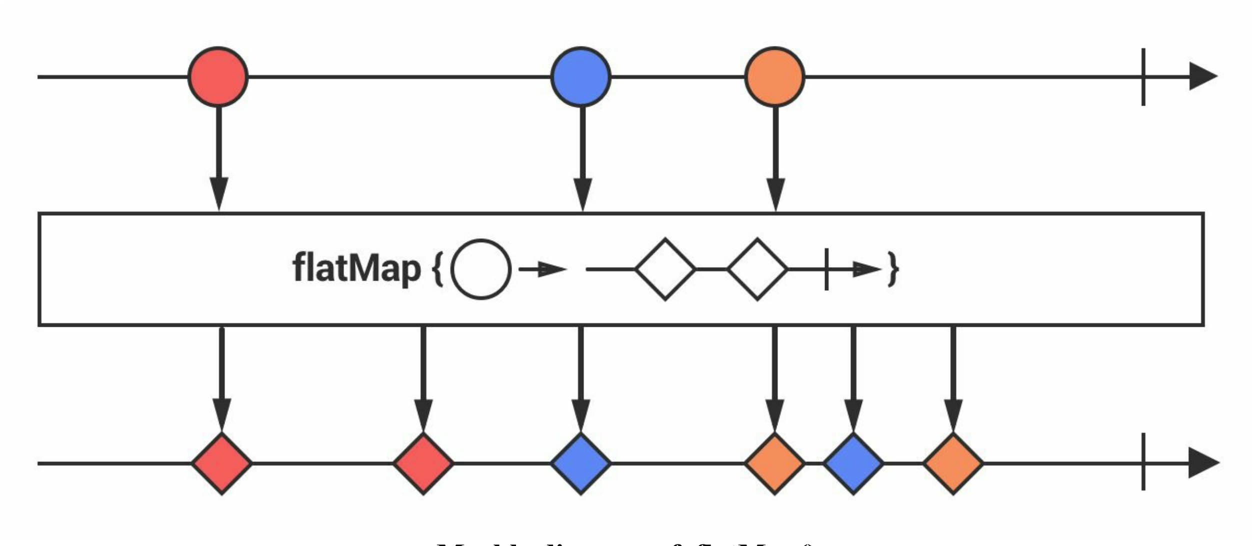
Something to note about using any `Operator` is that it is discouraged and non-idiomatic to mutate state internally and externally from the stream. For example, it is bad practice to do the following:

```java
List<String> allPostText = new ArrayList<>();
Observable<Post> postObservable = // ...
postObservable.map(post -> {
        String text = post.text;
        allPostText.add(text);
        return text;
    })
    .filter(text -> text.length() < 160)
    .subscribe(text -> {
        // Receive text that is less than 160 characters
    });
```

The reason this is discouraged is that mutating state internally and externally can have some unintended side-effects. Each `Operator` is designed to return a new `Observable` leaving the original `Observable` the way it was. This design makes reasoning simpler as it guarantees that each consumer is independent and any unique chain of `Operator`s applied by each one would not affect the other. There are a few cases where it is difficult to get around this; for example, if you had an `Observable<List<T>>` and wanted to sort each emission via `Collections.sort()`. These, however, should be treated as exceptions and avoided if possible.

FlatMap

`.flatMap()` is another very common `Operator` that is used to transform emissions from an `Observable`. The stream is transformed by applying a function that you specify to each emitted item. However, unlike `.map()`, the function specified should return an object of type `Observable`. The returned `Observable` is then immediately subscribed to and the sequence emitted by that `Observable` is then merged and *flattened* along with subsequent emissions which are then passed along to the observer.



Marble diagram of .flatMap()

A common real-world example of using `.flatMap()` is for implementing nested network calls. Say we have an `Observable<User>`, and for each `User`, we'd like to make a network call to retrieve that user's detailed information.

Assume we have the following methods available:

```
Observable<User> getUser(int userId) {
    // Return an Observable<User> that performs a network
    // request to retrieve a User with ID `userId`
}

Observable<UserDetail> getUserDetail(User user) {
    // Return an Observable<UserDetail> that performs a
    // network request to retrieve details about `user`
}
```

Chaining `.getUser()` with `.flatMap()` and having the function provided in `.flatMap()` return a `.getUserDetail()` gives us:

```
Observable<User> userObs = getUser(123);
Observable<UserDetail> detailObs = userObs.flatMap(user -> {
    return getUserDetail(user);
});
detailObs.subscribe(userDetail -> {
    // Received user's detail information here
});
```

Now you might still be wondering, what is the different between `.map()` and `.flatMap()` and when should you be using one over the other? Perhaps a good way to see the difference between the two is by changing our above example to use `.map()` instead.

```
Observable<User> userObs = getUser(123);
Observable<Observable<UserDetail>> detailObs = userObs.map(user -> {
    return getUserDetail(user);
});
detailObs.subscribe(userDetailObservable -> {
    // Received Observable<UserDetail> here, need to subscribe
    // to it to receive its emission
    userDetailObservable.subscribe(userDetail -> {
        // Got user detail
    });
});
```

Using `.map()`, the observer receives an object of type `Observable<UserDetail>` and thus needs to subscribe to it to receive its emissions. With `.flatMap()`, this is not the case; the observer receives a `UserDetail`, which, in our example, is the result the observer is interested in.

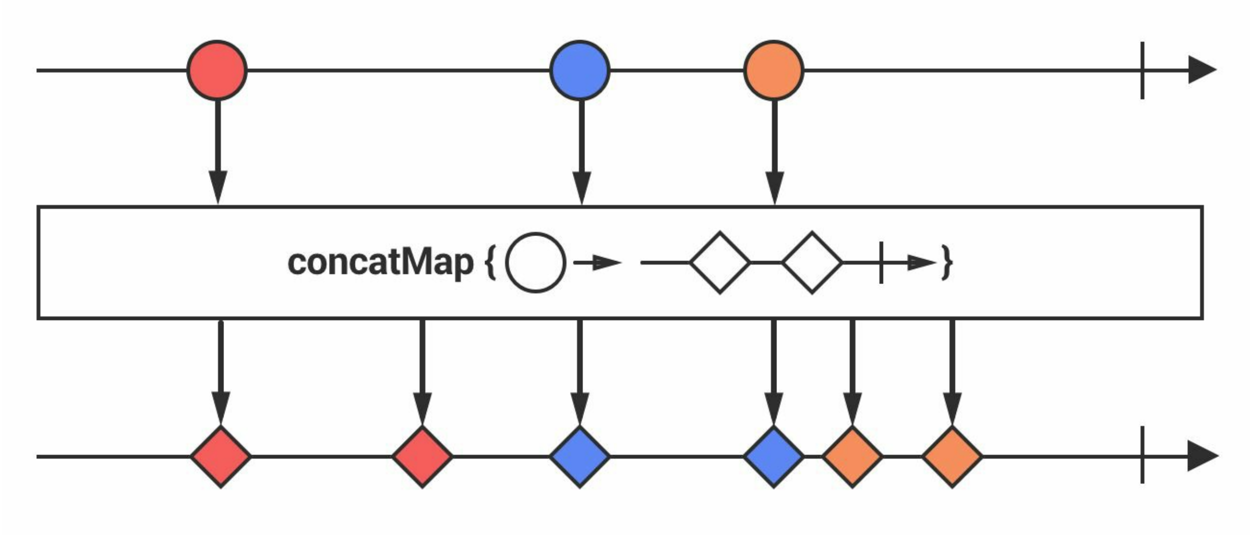
It is quite cumbersome for an observer to receive `Observable`s. Imagine for a second that our `.getUser()` method is implemented such that the observer receives updates to a `User` object (i.e. the `Observable` stream can return multiple objects over time). Using `.map()`, we would receive multiple `Observable<User>` objects which we would have to individually subscribe to and manage. This is exactly the problem that `.flatMap()` solves. The operator will handle managing `Observable` sequences emitted by the function you provide it so that the observer to `.flatMap()` receives a **flattened** stream.

Since each emission from `.flatMap()` can return a new sequence, this has several implications: (1) the returned `Observable` can have its own set of `Operator`s applied to it, and (2) if there is a delay between emissions, the emitted sequence can be interleaved with the sequence of other `Observable`s as depicted in the marble diagram. If this behavior of interleaving is not desired, `.concatMap()` can be used instead.

`.flatMap()` is also useful for introducing parallelism as we'll see in Chapter 4: Multithreading.

ConcatMap

`.concatMap()` transforms emissions by invoking a function that you specify. Similar to `.flatMap()`, the function you provide should also return an `Observable`. `.flatMap()` emissions, however, can be interleaved because it will immediately subscribe to emitted `Observable`s from the function provided (see `.flatMap()` diagram for reference). `.concatMap()`, on the other hand, will not interleave emissions. Instead, it will wait for the previous `Observable` to complete before subscribing to subsequent ones.



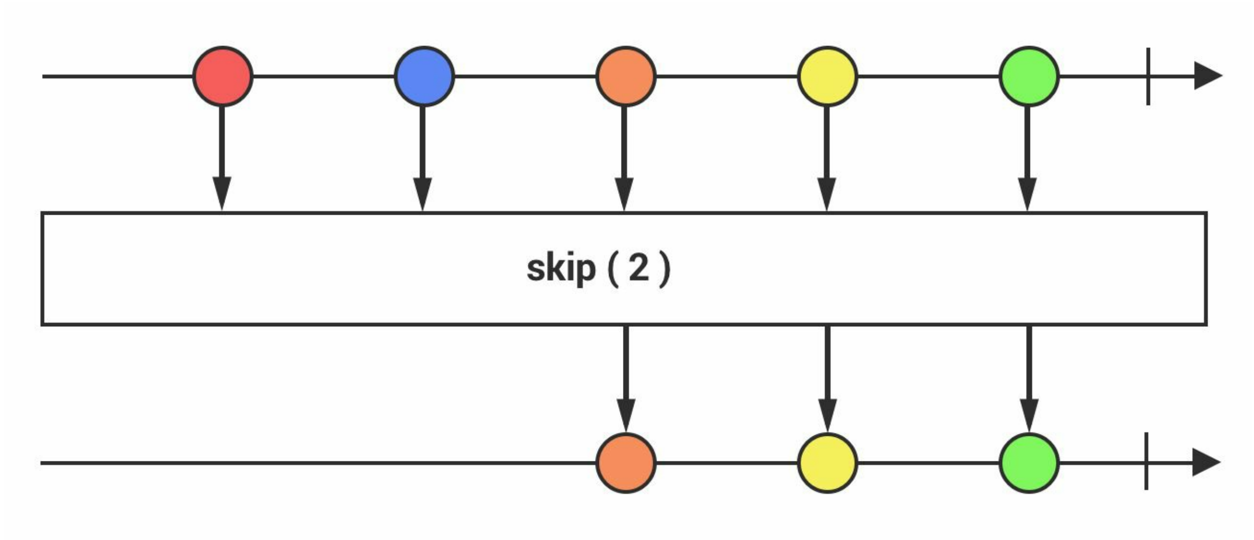
Marble diagram of .concatMap()

Note the performance implications of using `.concatMap()`—if a returned `Observable` from `.concatMap()` takes a while to complete, subsequent emitted `Observable`s will have to wait before subscribed to. This trade-off should be kept in mind when deciding to use `.flatMap()` vs. `.concatMap()`.

Skip

Other handy `Operator`s allow you to consume only a subset of the items emitted by the sequence.

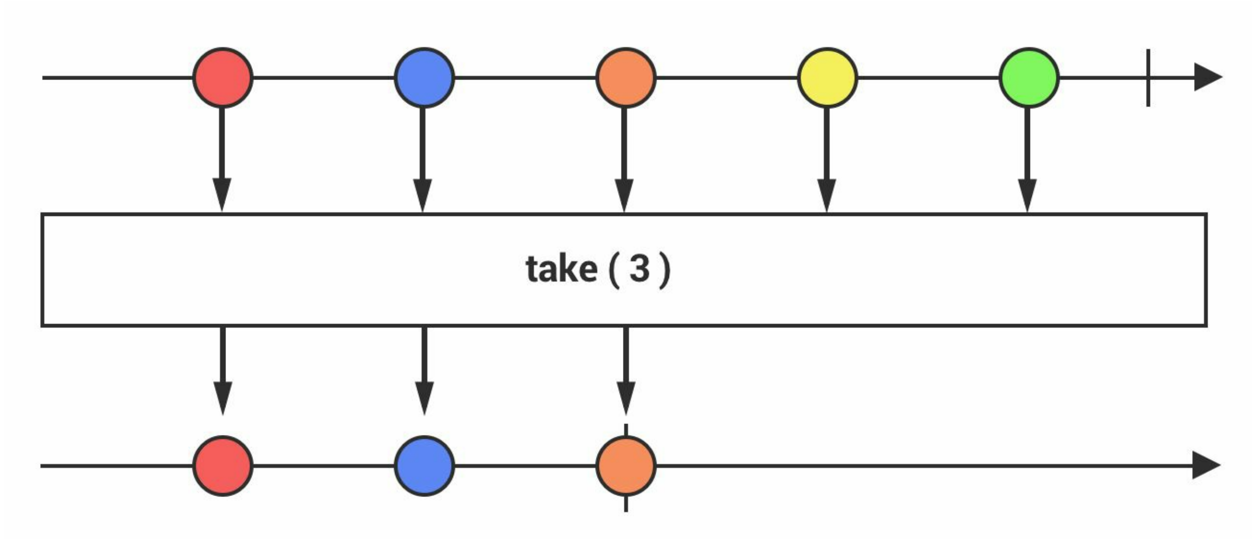
`.skip(n)` ignores the first `n` items emitted by the sequence and emits the remainder of the sequence. `.skip()` is also overloaded with `.skip(time, timeUnit)` which skips any item/s emitted before the specified time window.



Marble diagram of .skip()

Take

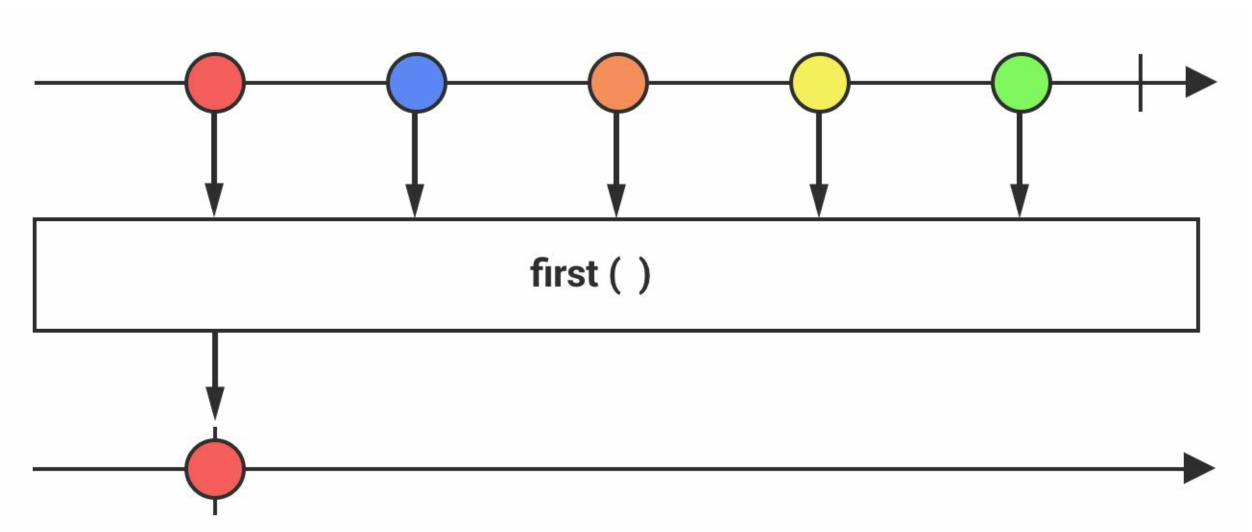
Mirroring `.skip()`, `.take(n)` takes the first `n` items emitted by the sequence and completes immediately after. Similarly, `.take(time, timeUnit)` only takes the item/s emitted in the specified time window and ignores the rest.



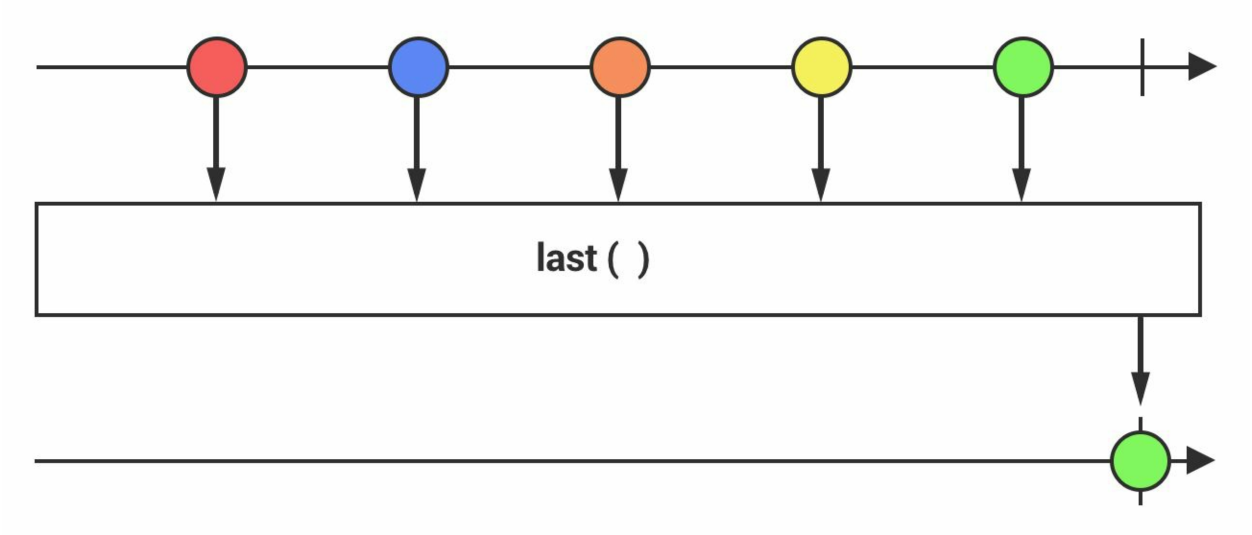
Marble diagram of .take()

First & Last

`.first()` & `.last()` are pretty self-explanatory. Note, however, that these methods return a `Single` rather than an `Observable`. A `Single` is a special subset of an `Observable` that emits just a single item rather than a sequence of items. It contains a subset of the `Operator`s that an `Observable` given its nature. We will look over `Single` in more detail in Chapter 5: Completable, Single, and Maybe



Marble diagram of .first()



Marble diagram of .last()

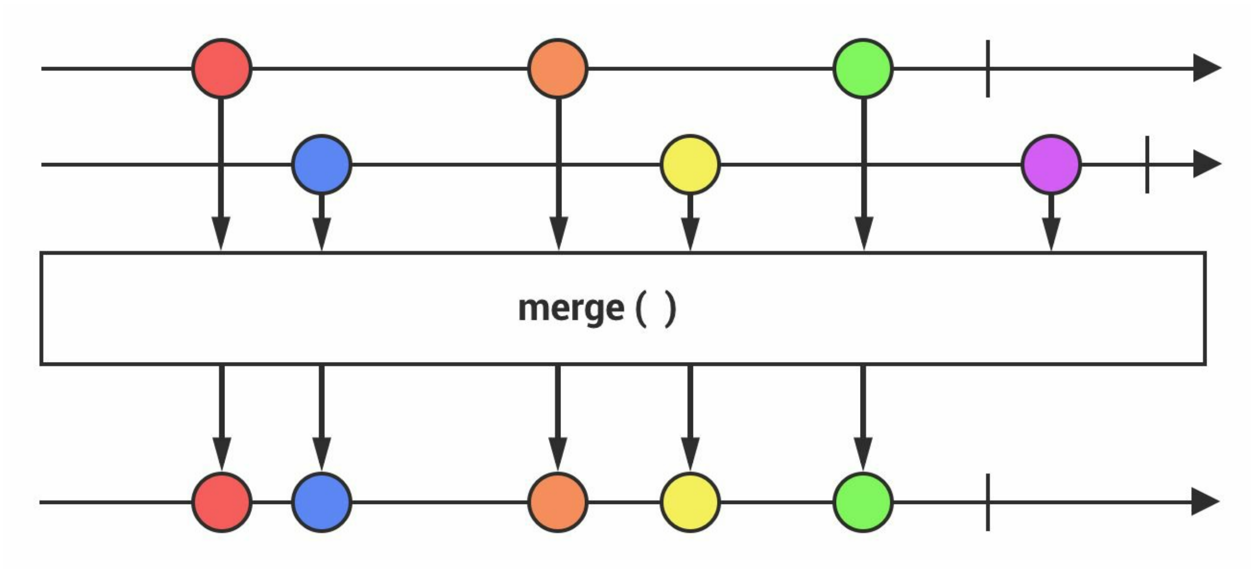
Notice for `.last()` that the item is not received immediately after it is emitted, it is received only after the upstream sequence completes.

Combining

Oftentimes you may find the need to combine several `Observable` streams into a single stream. Perhaps you have independent streams coming from multiple different sources and you would like to combine the results of these. Traditionally in Java, this task is especially difficult since synchronizing and blocking to wait on data coming from different threads of execution is error-prone and hard to debug. With RxJava, there are a few handy `Operator`s that allow you to combine several streams into a single unified stream.

Merge & Concat

There are several overloaded flavors of the `.merge()` `Operator`, all of which allow *merging* of multiple `Observable` streams into a single stream.



Marble diagram of .merge()

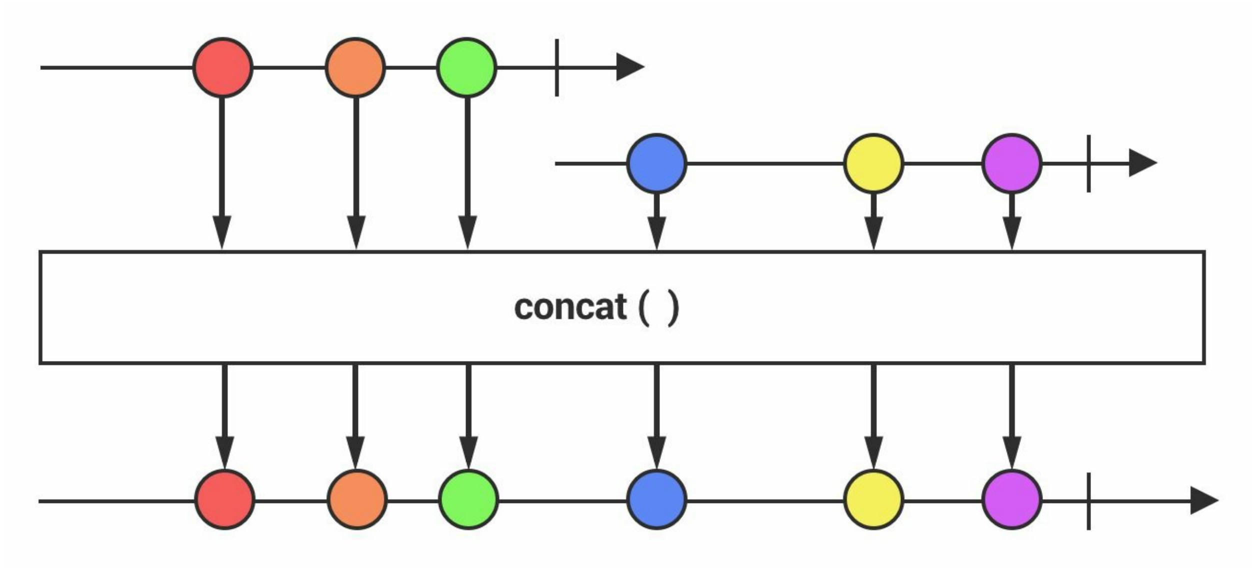
Say, for instance, we wanted to combine the results of places recommended by 2 different recommendation engines and merge the results of those two. If each recommendation was encapsulated in an `Observable`, combining the results of these would be a trivial operation with `.merge()`:

```
Observable<Place> getNearbyPlacesGoogle(Location location) {
    // Method that makes a network call to a Google API
    // to get all nearby places from `location`
}

Observable<Place> getNearbyPlacesFourSquare(Location location) {
    // Method that makes a network call to Four Square's API
    // to get all nearby places from `location`
}
```

Using the two methods defined for retrieving place recommendations, we can combine the results of each through `.merge()`:

```
Observable<Place> allNearbyPlaces = Observable.merge(
    getNearByPlacesGoogle(location),
    getNearByPlacesFourSquare(location)
);
allNearbyPlaces.subscribe(place -> {
    // Get all nearby places here
});
```

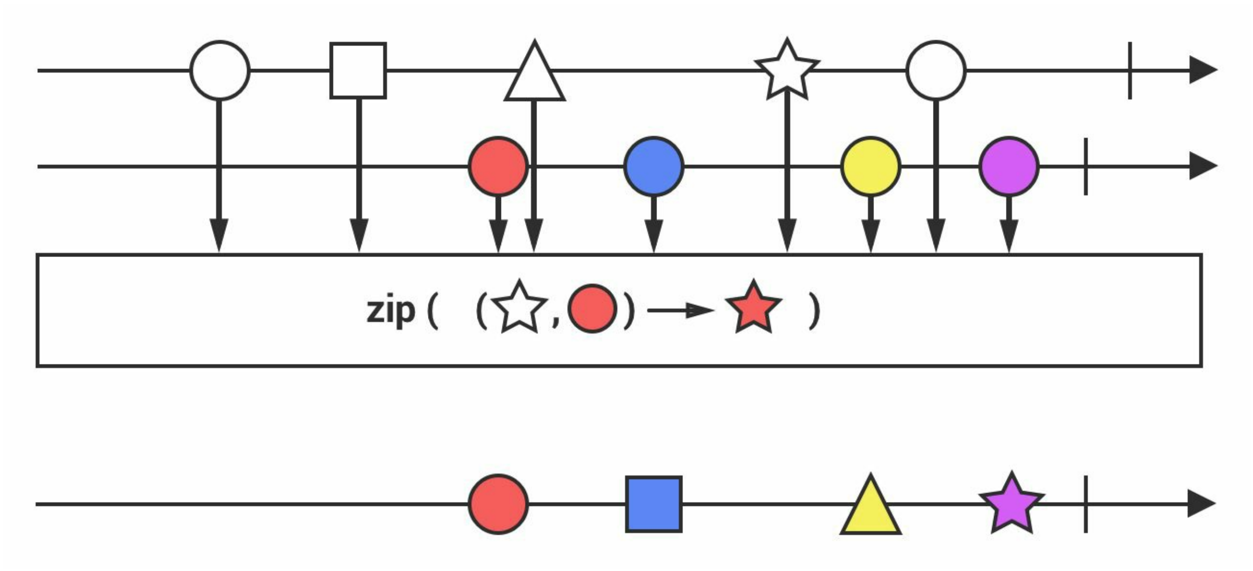
As soon as an `Observer` subscribes to the `Observable` **allNearbyPlaces**, each passed in `Observable` is individually subscribed to internally in `.merge()` and the result is then *merged* and combined. Note that from the marble diagram of `.merge()`, the resulting emissions can be interleaved, and once an `.onComplete()` is received from an `Observable`, the merged `Observable` completes as well. If the interleaving behavior is undesired, the `.concat()` `Operator` may be used to *concatenate* the results rather than *merge* them. In effect, the passed in `Observable` instances are subscribed to serially as each one completes rather than simultaneously. If the preceding `Observable` does not complete however (i.e. an infinite stream), the subsequent `Observable` will not be subscribed to.



Marble diagram of .concat()

Zip

`.zip()` enables pair-wise *zipping* of emissions from multiple different `Observable`s. It works by combining the emission from each `Observable` using the specified combiner function. This is especially handy when you have two data sources of different types that you would like to combine into a single `Observable` source.



Marble diagram of .zip()

Each zipped item is emitted in a strict sequence: the result of the 1st zipped item is the combination of the 1st item in each stream, the 2nd zipped item is the combination of the 2nd item in each stream, and so on. This implies that the zipped `Observable` will emit the same number of items emitted by the `Observable` that has the fewest items; if one sequence does not emit any items, the zipped `Observable` will also not emit any items.

> The size of `.zip()`'s output sequence is the same size as the source `Observable` with the smallest sequence. If one `Observable` does not emit any items, the zipped `Observable` will also not emit any items.

Say for example we wanted to make 2 separate network calls to populate a user's profile view. One network call to retrieve a user's detail information and another to get all the places they have been before:

```
Observable<UserDetail> getUserDetail(User user) {
    // Make network call to get `user`'s detailed information
}

Observable<List<Place>> getVisitedPlaces(User user) {
    // Make network call to get all the places `user` has visited.
}
```

Using `.zip()` followed by combining the result into a **android.support.v4.util.Pair** object (a generic wrapper that can hold two separate objects), we would get:

```
Observable.zip(
    getUserDetail(user),
    getVisitedPlaces(user),
    (userDetail, visitedPlaces) -> {
        return new Pair(userDetail, visitedPlaces);
}).subscribe(pair -> {
    UserDetail userDetail = pair.first;
    List<Place> visitedPlaces = pair.second;

    // Update profile view
});
```

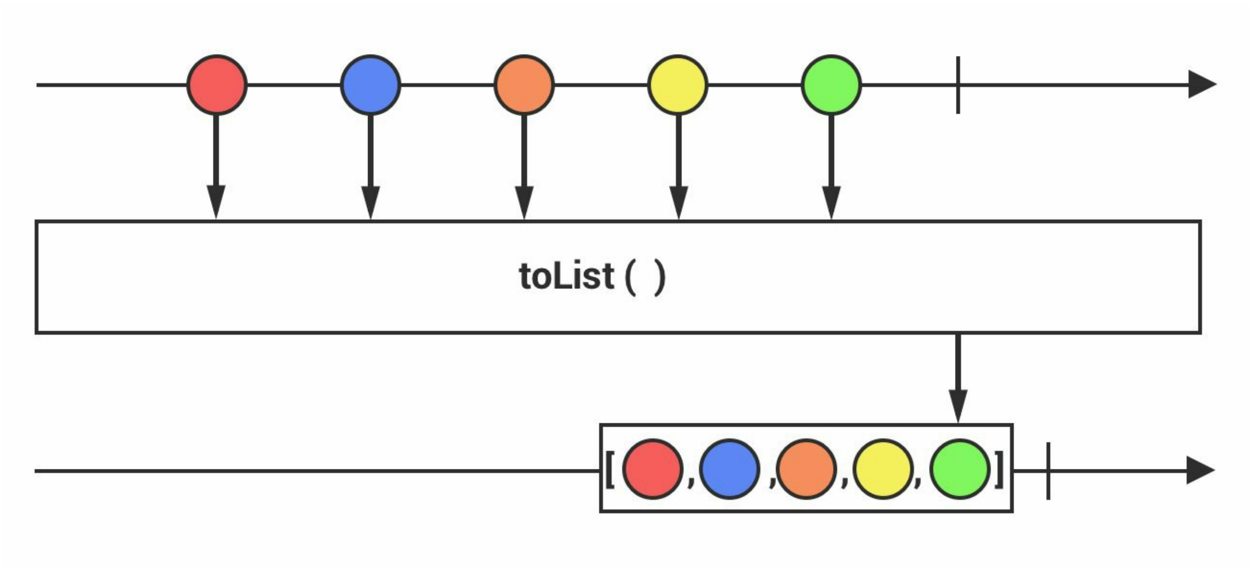
Another thing to note about `.zip()` is that if one sequence emits much faster than another, internally, `.zip()` will have to buffer the results of the faster `Observable` and wait until the slower `Observable` emits an item. We will revisit some more implications of this in Chapter 7: Backpressure.

Aggregating

RxJava has several `Operator`s that aggregate emissions from `Observable`s to provide `Observer`s with an intended result. `Operator`s that aggregate are typically stateful as internal state must be maintained between emissions. A few commonly used aggregation `Operator`s are `.toList()`, `.reduce()`, and `.collect()`.

ToList

`.toList()` converts an `Observable<T>` into an `Observable<List<T>>`. This is useful if the `Observer` ultimately cares about a `List<T>` but the stream itself pushes objects of type `T`.



Marble diagram of .toList()

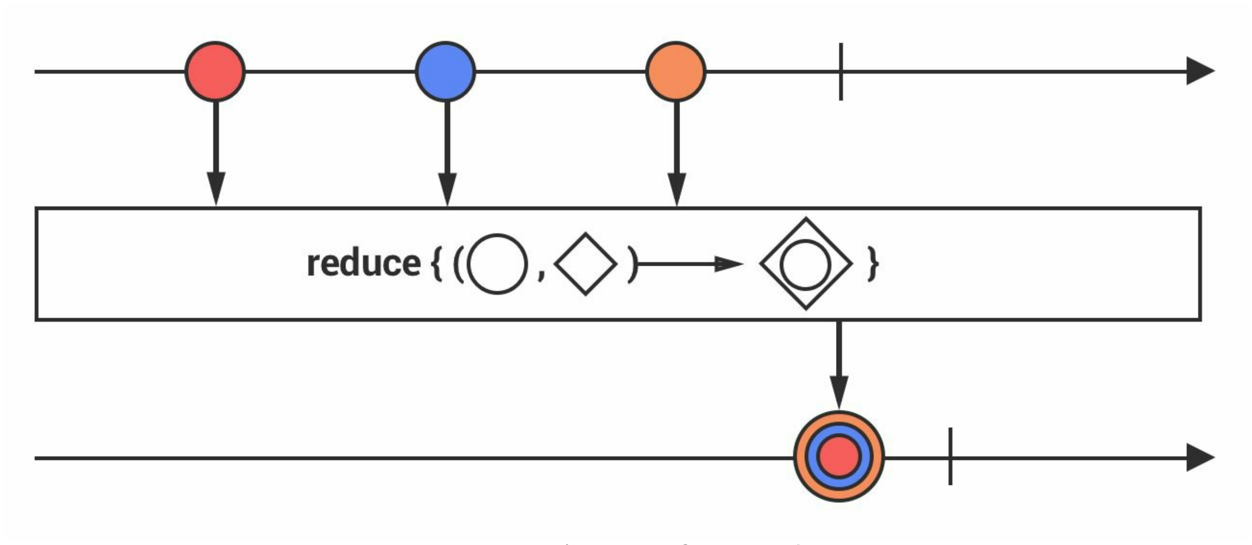
Notice in the marble diagram that `.toList()` will only emit the buffered list once upstream emits an `.onComplete()` notification. If `.onComplete()` is never invoked, `.toList()` will also never emit. This is a common source of bugs for beginners and should be used with caution.

Using our example for all nearby places, we chain a `.toList()` call to provide the `Observer` with a list of `Place` objects instead

```
Observable<Place> allNearbyPlaces = // ...
Observable<List<Place>> allNearbyPlacesList =
    allNearbyPlaces.toList()
        .subscribe(places -> {
            // receive a list of places here
        });
```

Reduce & Collect

`.reduce()` operates by applying an accumulator function on each emission to ultimately *reduce* the stream to a single emission.



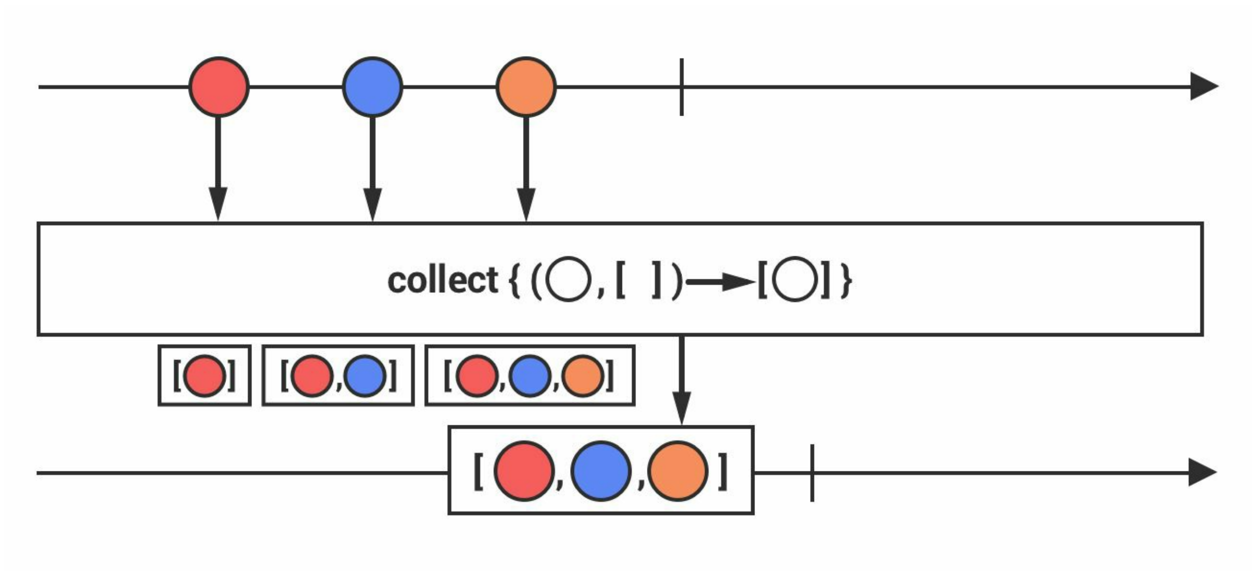
Marble diagram of .reduce()

For example, say we had a stream that emits integers and we are interested in computing the summation of those integers. For that, we can use `.reduce()` to accumulate the sum.

```
Observable<Integer> intStream = // ...
intStream.reduce(
    (i1, i2) -> i1 + i2
).subscribe(total -> {
    Log.d(TAG, "Total is: " + total);
});
```

In our example, `i1` is the summation of previous emissions, and `i2` is the current integer emission. The starting value for `i1` is `0`, but `.reduce()` also has an overloaded version where a different starting (seed) value may be specified. `.reduce()` however should not be used for accumulating data into a mutable data structure. For this, `.collect()` should be used.

`.collect()` operates very similarly to `.reduce()`, but instead of taking in an accumulator function, we pass it a `Callable` that emits the mutable data structure, followed by a `BiConsumer` that modifies the mutable data structure.



Marble diagram of .collect()

For example, say we had an `Observable` that emits a list of words. We can accumulate the list of words in a `StringBuilder`:

```
final Observable<String> wordStream = // ...
wordStream.collect(() -> {
    return new StringBuilder();
}, (stringBuilder, s) -> {
    stringBuilder.append(" ");
    stringBuilder.append(s);
}).subscribe(stringBuilder -> {
    Log.d(TAG, stringBuilder.toString());
});
```

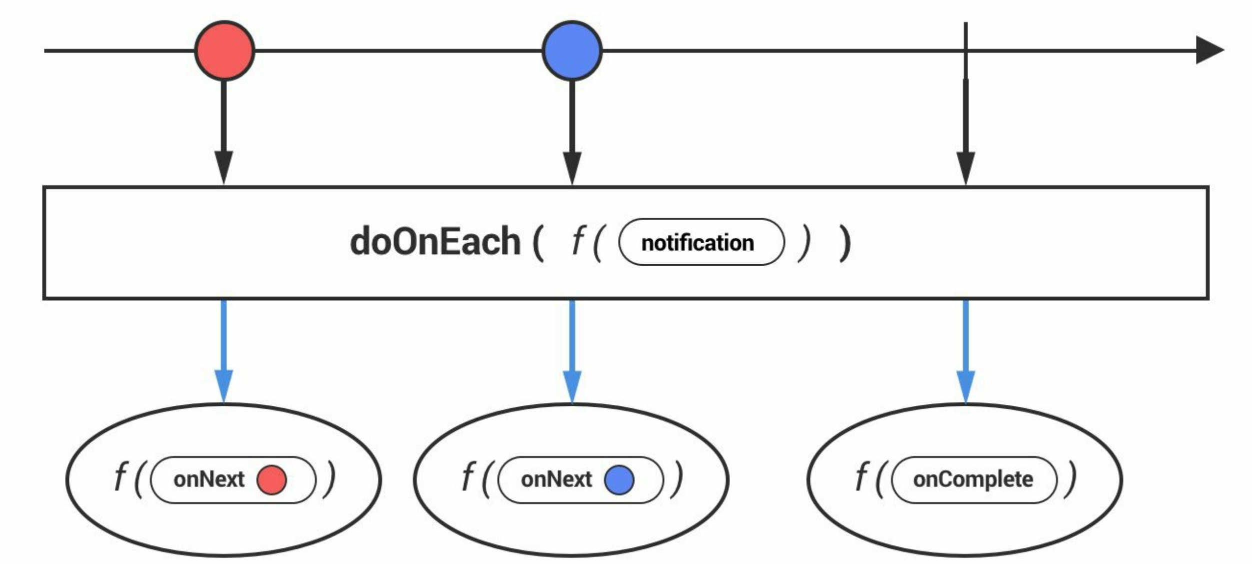
Utility Operators

RxJava has a handful of utility `Operator`s that don't necessarily modify the emissions themselves through transformations/filters, but instead allow you to do various actions such as getting insight into events in the stream itself (for debugging/logging purposes) or caching results emitted in the stream.

DoOnEach

`.doOnEach()` is an `Operator` that enables listening to events that occur in an `Observable` stream (i.e. when `.onNext()`, `.onError()` and `.onComplete()` are invoked) by providing an `Observer`. Generally, this `Operator` is used if you would like to perform a side-effect when an event occurs. There are also several `.doX()` `Operator`s–such as `.doOnNext()`, `.doOnError()` and

`.doOnComplete()`–that are available if you're interested in listening only to a specific event.



Marble diagram of .doOnEach()

Some common usages for `.doX() Operator`s is for logging purposes. Say we had a chain of operators in a stream and we would like to log each time an `Operator` is about to be applied.

```
Observable<Integer> intStream = Observable.range(1, 3);
intStream.doOnNext(i -> Log.d(TAG, "Emitted: " + i))
    .map(i -> i * 2)
    .doOnNext(i -> Log.d(TAG, "map(): " + i))
    .filter(i -> i % 2 == 0)
    .doOnNext(i -> Log.d(TAG, "filter(): " + i))
    .subscribe(i -> {
        Log.d(TAG, "onNext(): " + i);
    });
```

As expected, the following code snippet would log on the console the result of each operation (new lines added for readability).

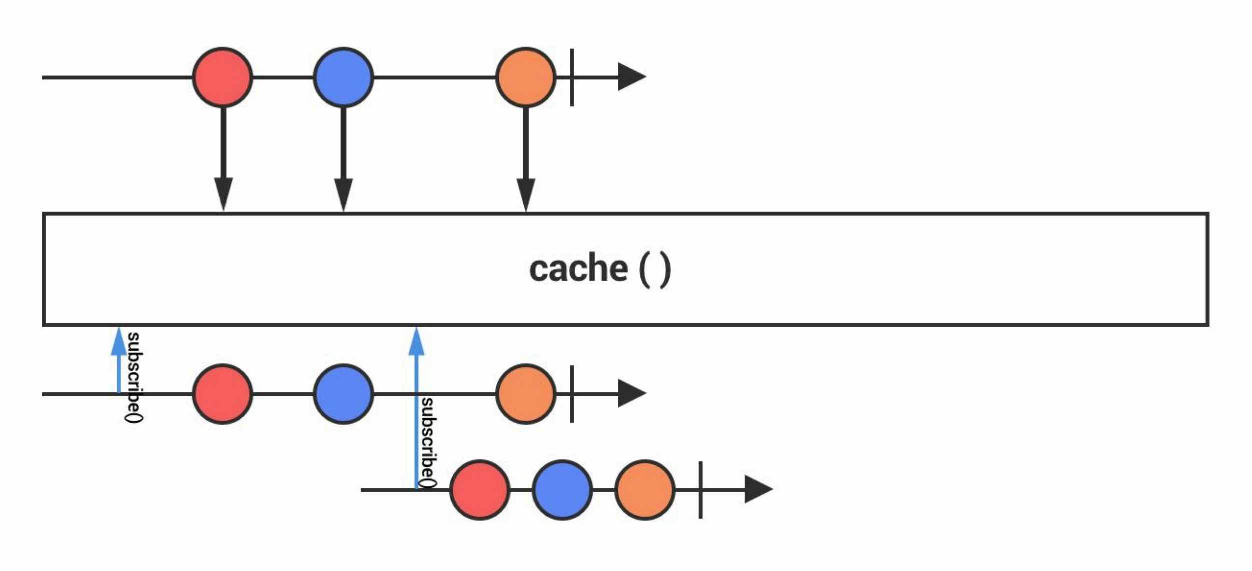
```
Emitted: 1
map(): 2
filter(): 2
onNext(): 2

Emitted: 2
map(): 4
filter(): 4
onNext(): 4

```

```
Emitted: 3
map(): 6
filter(): 6
onNext(): 6
```

Of course logging can also be performed within the `Operator`s themselves, however, this should be avoided as it doesn't stay true to the philosophy of maintaining single/simple purposed `Operator`s.

Cache

`.cache()` is a handy `Operator` that subscribes lazily to an `Observable` and caches all of the events emitted in the `Observable` so that when another `Observer` subscribes at a later point, the cached events will be replayed. `.cache()` can be used when you would like to reuse events that were already computed versus having to recompute the events all over again.



Marble diagram of .cache()

Say you need to make a network call to obtain app-specific settings (e.g. feature flags, server-generated strings, etc.). This network call needs to be made when the app first comes to the foreground and the result of which should be reused throughout the app. One way to solve this is by using `.cache()` and reusing the resulting `Observable`'s initial emission.

```
Observable<Timed<AppSettings>> appSettingsObservable =
    Observable.fromCallable(() -> {
        return apiService.getAppSettings();
    }).timestamp().cache();

// Make 1st network call when app comes to foreground.
appSettingsObservable.subscribe(result -> {
    Log.d(TAG, "Time computed on foreground: " + result.time());
});

// Get settings at a later point in the app
appSettingsObservable.subscribe(result -> {
    Log.d(TAG, "Time computed cached: " + result.time());
});
```

In addition to `.cache()`, we've also added a `.timestamp()` call which is another utility `Operator` that wraps an emission in a `Timed` object–a class that wraps an arbitrary object (in this case, the retrieved `AppSettings`) along with a timestamp of when the item was emitted. Running this, we would verify that the `AppSettings` was indeed reused rather than recomputed.

```
Time computed on foreground: 1488282552026
Time computed cached: 1488282552026
```

Reusing Operator Chains

As mentioned earlier in Chapter 2, using an `Operator` should be kept single-purposed. If multiple transformations need to be applied to an emission, each transformation should be encapsulated in its own `Operator`. This approach is declarative (i.e. it describes *what* should be done, not *how* it should be done) which aids in readability.

As such, you might find yourself repeating the same chain of `Operator`s in parts of your codebase. For example, say it was a very common action in one of your classes to filter by some condition, skip the first filtered item, followed by taking the first result after skipping:

```
someObservable.filter(this::isConditionSatisfied)
    .skip(1)
    .take(1)
    // ...continue with other operations here
```

In the spirit of the DRY (i.e. don't repeat yourself) principle, these repeated operator chains can be shared and reused by using an `ObservableTransformer` and passing that as a parameter to the `.compose()` operator.

In addition to `ObservableTransformer`, other `XTransformer` classes exist for transforming other base reactive types (see Chapter 5: Completable, Single and Maybe).

ObservableTransformer

An `ObservableTransformer` allows us to compose/transform `Observable`s. In our example above, we can encapsulate the three operators–namely, `.filter()`, `.skip()` and `.take()`–inside of an `ObservableTransformer` and reuse it across other `Observable`s.

```
public class MyTransformer <T> implements ObservableTransformer<T, T> {

    private final Predicate<? super T> filterVal;
    private final int skipVal;
    private final int takeVal;

    MyTransformer(
        Predicate<? super T> filterVal,
        int skipVal,
        int takeVal
    ) {
        this.filterVal = filterVal;
        this.skipVal = skipVal;
        this.takeVal = takeVal;
    }

    @Override
    public ObservableSource<T> apply(Observable<T> upstream) {
        return upstream.filter(filterVal)
                       .skip(skipVal)
                       .take(takeVal);
    }
}
```

The class `MyTransformer` implements `ObservableTransformer` and has the overridden method `.apply()` where we are passed in the source `Observable`. Within it, we apply the necessary transformations and return the resulting `Observable`.

To use this, we would create an instance of `MyTransformer` and pass it via `.compose()`:

```java
someIntObservable.compose(
    new MyTransformer<Integer>(this::isConditionSatisfied, 1, 1)
)
    // ...continue with other operations here
```

As you can see, using `.compose()` along with an `ObservableTransformer` promotes code reuse, and if any changes are needed in the operator chain, we would simply update the operator chain inside `MyTransformer`.

Now, what about code reuse by using a plain-old Java method and calling it from within an `Operator`? Technically that can also work, but it all depends on the operation to be applied. For example, if you need to transform *every single* emission in a stream, we can simply call a method from within a `.map()` operator:

```java
someIntObservable.map(this::transform)
```

However, in our previous example, we are not applying a transformation for every single emission. Instead, we are filtering the stream by some condition and taking the 2nd result from that filter. In other words, we need to transform *the stream itself* to obtain the result we are interested in. With `.compose()`, we can do just that and reuse multiple `Operator`s together.

Chapter 4: Multithreading

From the previous chapters, we have looked at the `Observable` and the `Observer` pair along with several `Operator`s that can modify emissions from an `Observable` stream. So far we have avoided dealing with any sort of concurrency; however, one of the hallmarks of RxJava is that it simplifies working with concurrent tasks. While traditional Java/Android threading options such as `Thread`s, `Executor`s, and `AsyncTask`s are still available and can be used in conjunction with RxJava, RxJava offers its own threading model that works well within the stream abstraction. In this chapter, we will go through these constructs and show you how you can control which thread each computation should occur on.

Asynchronicity

It is a common misconception that RxJava is inherently asynchronous–perhaps the reason for this confusion is because `Observable`s are *push* instead of *pull* by nature. But, in fact, `Observable` streams are synchronous by default unless otherwise specified.

Given that `Observable` streams are synchronous by default, can you guess the order of the Logcat statements in the following example…?

```
Observable<Integer> integerObservable = Observable.create(source -> {
    Log.d(TAG, "In subscribe");
    source.onNext(1);
    source.onNext(2);
    source.onNext(3);
    source.onComplete();
});
Log.d(TAG, "Created Observable");

Log.d(TAG, "Subscribing to Observable");
integerObservable.subscribe(i -> Log.d(TAG, "In onNext(): " + i));

Log.d(TAG, "Finished");
```

Unsurprisingly, the above code will print out the following:

```
Created Observable
Subscribing to Observable
In subscribe
In onNext(): 1
In onNext(): 2
In onNext(): 3
Finished
```

As soon as `.subscribe()` was invoked, control was passed to the `Observable` to generate and emit its data stream. In other words, the `.subscribe()` call causes the `Observable` to do the work specified in the function that was passed to the `.create()` method. But what happens if this work is a long-running operation (which in many real-world cases, it would be)? If it is invoked on Android's main UI thread, then we have problems… namely, user input being blocked leading to the dreaded "Application Not Responding" (ANR) dialog.

Adding Asynchronicity

There are many ways we can add asynchronicity so that the function within `.create()` is invoked on a separate thread from the calling thread. As mentioned in Chapter 2, RxJava is agnostic as to how asynchronicity is accomplished. One way is to make use of a plain old `Thread`. Let's take our previous example… instead of immediately calling `.onNext()` in the function provided in `.create()`, we will…

1. … wrap the `.onNext()` and `.onComplete()` calls in a `Runnable`,
2. … provide the `Runnable` to a new `Thread`,
3. … and then start the Thread.

```java
Observable<Integer> integerObservable = Observable.create(source -> {
    Log.d(TAG, "In subscribe");
    new Thread(() -> {
        source.onNext(1);
        source.onNext(2);
        source.onNext(3);
        source.onComplete();
    }).start();
});
Log.d(TAG, "Created Observable");

Log.d(TAG, "Subscribing to Observable");
integerObservable.subscribe(i -> Log.d(TAG, "In onNext(): " + i));

Log.d(TAG, "Finished");
```

With this change, we now get:

```
Created Observable
Subscribing to Observable
In subscribe
Finished
In onNext(): 1
In onNext(): 2
In onNext(): 3
```

As we can see, the "Finished" statement is invoked before the "onNext()" statements. In fact, the order of these statements are no longer deterministic and our original log sequence is still possible; it all depends on when the OS decides to perform a context switch, which can change from one execution to the next.

This approach to non-blocking subscription is perfectly valid as long as an invocation to `.onNext()`, `.onError()`, or `.onComplete()` is performed from a single thread and that it is thread-safe. If this contract is breached, some `Operator`s that depend on this assumption may fail to function (i.e. we cannot emit events from multiple threads).

The above example was used to show that a plain old `Thread` (and any other Java/Android threading construct) can be used to add asynchronicity to RxJava. However, just because using these is *possible* does not mean that they *should* be used. The preferred way of achieving concurrency in RxJava is actually through the use of `Scheduler`s.

Schedulers

A `Scheduler` is a multithreading construct introduced in RxJava that can run a scheduled unit of work on a thread. Simplistically, you can think of a `Scheduler` as a thread pool, and when a task needs to be executed, it takes a single thread from its pool and runs the necessary task. `Scheduler`s are used in conjunction with `.subscribeOn()` and `.observeOn()`, two `Operator`s that specify where a particular operation should execute. When a `Scheduler` is specified along the `Observable` chain, the `Scheduler` will provide a worker that will run the actual block of code. The thread that the scheduled work executes on depends on the `Scheduler` in use. There are several types of `Scheduler`s available via factory methods in the `Schedulers` class:

- **Schedulers.newThread()**, as its name suggests, returns a `Scheduler` that will create a new `Thread` for each scheduled unit of work. As a result, using `Scheduler.newThread()` is expensive since a new `Thread` is spawned each time–no reuse happens.
- **Schedulers.from(Executor executor)** returns a `Scheduler` that is backed by the specified `Executor`.
- **Schedulers.single()** returns a `Scheduler` that is backed by a single thread. If several tasks are scheduled in `Schedulers.single()`, the tasks are scheduled serially and will be executed in order.
- **Schedulers.io()** provides a `Scheduler` that is meant for I/O-bound work such as network calls, database transactions, etc. The underlying implementation uses an `Executor` thread pool that will grow as needed. With I/O tasks, we are not doing CPU-intensive work; instead, we are usually waiting on resources or services that may live on a different host, so being able to create and assign a new thread to each task is ideal.
- **Schedulers.computation()** provides a `Scheduler` that is meant for computational or CPU-intensive work such as processing a long list, resizing an image, etc. This `Scheduler` utilizes a thread pool whose size is bounded by the number of available cores. Because we are performing CPU-intensive work, creating more threads than cores will actually degrade performance; both thread creation and context switching result in a lot of overhead without helping to speed up our CPU-intensive work. The `.computation() Scheduler` should never be used for I/O, and the `.io() Scheduler` should never be used for computational work; I/O and computation block for completely different reasons thereby requiring completely different threading strategies.
- **Schedulers.trampoline()** provides a `Scheduler` that schedules work on the current thread (i.e. using this `Scheduler` will block the current thread). Scheduled tasks are not executed immediately and instead are put in a queue and executed after the current unit of work is completed. This particular `Scheduler` is typically used when implementing recursion to avoid infinitely growing the call stack.

Observable.subscribeOn()

The `.subscribeOn()` operator is used in the `Observable` chain to dictate where the `Observable` should operate (i.e. the function inside of `.create()`). Rewriting the previous example using a `Scheduler` instead gives us:

```
Observable<Integer> integerObservable = Observable.create(source -> {
    Log.d(TAG, "In subscribe");
    source.onNext(1);
    source.onNext(2);
    source.onNext(3);
    source.onComplete();
});
Log.d(TAG, "Created Observable");

Log.d(TAG, "Subscribing to Observable");
integerObservable.subscribeOn(Schedulers.newThread())
                 .subscribe(i -> Log.e(TAG, "In onNext(): " + i));

Log.d(TAG, "Finished");
```

Running the above code is similar to using a `Thread` in that the operations inside `.create()` will now occur in a separate thread provided by `Schedulers.newThread()`. The benefit of this approach over using a `Thread` is that tacking on a `Scheduler` to specify where the `Observable` should execute is *declarative* and we no longer have to worry about the low-level details of how to create and run a thread. Furthermore, we can decouple the `Observable` creation from the threading; the subscriber can determine the thread on which to run the `Observable` rather than having that decision baked into the `Observable` itself.

> `.subscribeOn()` controls what `Scheduler` is used to execute the statements inside the `.create()`.

Observable.observeOn()

Using `.subscribeOn()`, we are able to modify where an `Observable` does its work. But say, for example, that we performed a network call to retrieve a `User`'s information using `Schedulers.io()` and we want to update the UI once the network call succeeds. Can you spot the bug in the following line of code?

```
// `userObservable` retrieves a User object over the network
Observable<User> userObservable = // ...

```

```
userObservable.subscribeOn(Schedulers.io())
              .subscribe(user -> textView.setText(user.getName()));
```

As you might have noticed in the preceding example, all subsequent operations after `.subscribeOn()` will run on the thread provided by the specified `Scheduler`. In other words, the call to `textView.setText(user.getName())` will run on a thread other than the UI thread, which will cause the app to crash with a `CalledFromWrongThreadException`. To circumvent this issue, we can wrap the call to update the `TextView` in a `Runnable` and have it run on the main thread; however, doing that is about as elegant as our plain old `Thread` example above. Ideally, we want a declarative approach to switch threads and post emissions to a separate thread (in this case, to the Android main thread). For this, we make use of the `.observeOn()` operator.

`.observeOn()` modifies the `Observable` so that its emissions will be posted in the provided `Scheduler`. `.observeOn()` is the `Operator` we want if we want the downstream consumer to operate on a separate thread. To fix the code above, we would simply add an `.observeOn()` before the `.subscribe()` call.

```
// `userObservable` retrieves a User object over the network
Observable<User> userObservable = // ...

userObservable.subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(user -> textView.setText(user.getName()));
```

Running the above code will now execute `textView.setText(user.getName())` in the Android main thread. Note that the `Scheduler` we used is `AndroidSchedulers.mainThread()`, which is provided by the RxAndroid extension library to RxJava.

Scheduler Behavior Details

As you can see, using `.subscribeOn()` and `.observeOn()` make threading trivially easy, but there are some intricacies to these operators that need to be called out.

As pointed out earlier, where in the operator chain the `.subscribeOn()` call appears does not matter. However, it's important to note that only the *first*

`.subscribeOn()` call will be applied; any subsequent `.subscribeOn()` calls will have no effect. Oftentimes, `Observable`s are created as part of a library or API and provided to a client via some interface. This interface documentation should clearly state if the returned `Observable` will have already been relegated to some `Scheduler` via `.subscribeOn()` because, if it has, the client will no longer be able to make its own `.subscribeOn()` call. In general, it is recommended that the creator of the `Observable` *not* call `.subscribeOn()` to allow for flexibility by the client. For example, the client may already be operating on a background thread and has no need for the `Observable` to create yet another thread to perform its work. If, for some reason, you do *not* want the caller to be able to modify the `Scheduler` where the `Observable` executes, simply add a `.subscribeOn()` call to the `Observable` that you pass back.

> Only the first `.subscribeOn()` call—-no matter where it is in the operator chain—-will be applied to the source `Observable` at the time `.subscribe()` is called; any subsequent `.subscribeOn()` calls will have no effect.

> It is advised for `.subscribeOn()` to be placed first in the operator chain. Doing this will be clearer for a human reader since `.subscribeOn()` applies to the start of data stream.

`.observeOn()`, on the other hand, is the dual of `.subscribeOn()` in many ways. Whereas `.subscribeOn()` affects the upstream source `Observable`, `.observeOn()` affects the downstream operators. And whereas only the first `.subscribeOn()` takes effect, `.observeOn()` can be called multiple times, with each call switching the thread for downstream operators (until another `.observeOn()` call is encountered). For common use cases (e.g. where the `Observer` is updating the UI), it is recommended to put the `.observeOn()` call as close to the `.subscribe()` call (i.e. as late in the `Operator` chain) as possible. This will ensure the computation performed by the `Operator`s does not occur on the main UI thread, which we generally want to keep unburdened. And since thread switches bear some performance cost, we also don't want to perform them early on in the stream for items that may just get filtered out later on.

To reinforce the previous points, let’s take a look at a completely-non-real-world example. To help show which thread each operator is running on, let’s define the following `print()` method:

```java
public static void print(String message) {
    String threadName = Thread.currentThread().getName();
    Log.d(TAG, threadName + " : " + message);
}
```

And taking our `Observable` from before, let’s add a series of `.subscribeOn()` and `.observeOn()` calls to the chain to see what happens:

```java
 1 Observable<Integer> observable = Observable.create(source -> {
 2     print("In subscribe");
 3     source.onNext(1);
 4     source.onNext(2);
 5     source.onNext(3);
 6 });
 7
 8 observable.subscribeOn(Schedulers.computation()) // "RxComputationThreadPool-
 9     .doOnNext(value -> print("(a) : " + value))
10     .observeOn(Schedulers.newThread()) // "RxNewThreadScheduler-2"
11     .doOnNext(value -> print("(b) : " + value))
12     .observeOn(Schedulers.newThread()) // "RxNewThreadScheduler-3"
13     .subscribeOn(Schedulers.newThread()) // This has no effect.
14     .doOnNext(value -> print("(c) : " + value))
15     .observeOn(Schedulers.newThread()) // Overwritten by next observeOn().
16     .observeOn(AndroidSchedulers.mainThread()) // "main"
17     .subscribe(value -> print("(d) : " + value));
18 }
```

This code prints out the following output:

```
RxComputationThreadPool-1 : In subscribe
RxComputationThreadPool-1 : (a) : 1
RxComputationThreadPool-1 : (a) : 2
RxComputationThreadPool-1 : (a) : 3
RxNewThreadScheduler-2 : (b) : 1
RxNewThreadScheduler-2 : (b) : 2
RxNewThreadScheduler-2 : (b) : 3
RxNewThreadScheduler-3 : (c) : 1
RxNewThreadScheduler-3 : (c) : 2
RxNewThreadScheduler-3 : (c) : 3
main : (d) : 1
main : (d) : 2
main : (d) : 3
```

- Notice in the above code that the thread specified in the `.subscribeOn()` call on Line 8 (*RxComputationThreadPool-1*) applies

to both the source Observable as well as subsequent operators until Line 10, when a new thread (*RxNewThreadScheduler-2*) is specified by the `.observeOn()` call.
- Each `.observeOn()` call then changes the thread that the stream runs on until the next downstream `.observeOn()` call.
- Lastly, notice that the `.subscribeOn()` call on Line 13 has no effect on the output; there would be no difference if we had inserted that call anywhere from Line 9 to Line 16 or removed it completely; it's made completely irrelevant because of our earlier `.subscribeOn()` call on Line 8.

Concurrency with FlatMap

So far, we have looked at ways to introduce multithreading through the use of `Schedulers` along with the `.subscribeOn()` and `.observeOn()` operators. To achieve true concurrency, however, these operators alone are not sufficient.

If you noticed in our example of using `.subscribeOn()`, events received from upstream were not processed on separate threads–all emissions were executed on the same thread (i.e. *RxCachedThreadScheduler-1*). To achieve true concurrency, we would need the help of `.flatMap()`.

Say, for example, we wanted to get the corresponding `User` object given a list of usernames.

```
class User {
    final String username;
    // Other user fields go here...

    User(String username) {
        this.username = username;
    }
}

class NetworkClient {
    Random random = new Random();

    User fetchUser(String username) {
        // perform blocking network call here but for the sake of the example
        // mimic network latency by adding a random thread sleep and return
        // a new User object
        randomSleep();
        return new User(username);
    }

    void randomSleep() {
        try {
            Thread.sleep(random.nextInt(3) * 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Using the defined classes, we can make the network calls to fetch the `User` objects:

```
NetworkClient networkClient = new NetworkClient();
String[] usernames = new String[] {
    "john", "mike", "jacob"
};
Observable.fromArray(usernames)
    .subscribeOn(Schedulers.io())
    .map(networkClient::fetchUser)
    .subscribe(user -> print("Got user: " + user.username));
}
```

The first two lines initialize the `NetworkClient` object as well as a `usernames` String array. We then convert `usernames` into an `Observable<String>` using the `Observable.fromArray()` creation method. The `.map()` operator then retrieves the `User` object given a username which is then emitted to the observer.

Running this code would give us:

```
RxCachedThreadScheduler-1 : Got user: john
RxCachedThreadScheduler-1 : Got user: mike
RxCachedThreadScheduler-1 : Got user: jacob
```

Looking at the order of the printed statements, each `User` is fetched in the same order that it appears in the `usernames` array. The operation in `.map()` (i.e. `.fetchUser()`) will block until it completes before it can process another username. However, there is no reason that we need to wait for the network call to complete before retrieving another `User` object. All of these network requests can run concurrently.

As we've seen in the previous chapter, using `.flatMap()`, we are able to return a new `Observable` given an upstream emission; the returned `Observable` from `.flatMap()` is completely independent and we can even specify a separate `Scheduler` for that `Observable` to operate in. In this way, we can impose concurrent behavior in an `Observable` chain.

Modifying the code above to use `.flatMap()` instead of `.map()`, we get:

```
Observable.fromArray(usernames)
    .subscribeOn(Schedulers.io())
    .flatMap(username ->
        Observable.fromCallable(() ->
            networkClient.fetchUser(username)
        ).subscribeOn(Schedulers.io())
    )
    .subscribe(user -> print("Got user: " + user.username));
}
```

Running the above code gives us:

```
RxCachedThreadScheduler-3 : Got user: jacob
RxCachedThreadScheduler-2 : Got user: mike
RxCachedThreadScheduler-1 : Got user: john
```

Notice that the operations to fetch a `User` were now run on separate threads (i.e. *RxCachedThreadScheduler-3*, *RxCachedThreadScheduler-2* and *RxCachedThreadScheduler-1*), essentially unblocking execution of `User` fetches. As a side-effect of introducing concurrency, however, the order of emissions from `.flatMap()` may no longer be the same as the order of events received upstream.

Part 2: RxJava Advanced

Chapter 5: Reactive Modeling on Android

By now you should have a good understanding of the basic concepts of RxJava. We covered the building blocks which are composed of the 3 O's: the `Observable`, the `Observer`, and the `Operator`. We've also looked at several functional style operations in the RxJava toolkit that transform, combine and aggregate emissions from an `Observable` until we received a desired result. You should now also have a pretty good sense of how RxJava can simplify dealing with concurrent processes inherent in developing mobile apps through the use of `Scheduler`s. Collectively, these tools allow us to program in Android in a way that is declarative over the traditional imperative approach.

In this chapter, we will combine all the lessons learned so far and go over some examples of what can be modeled from the non-reactive Android world into the reactive world. We will also go over a few new concepts that will ultimately aid in making the transition to a reactive codebase.

Bridging Non-Reactive into the Reactive world

One of the challenges of adding RxJava as one of the libraries to your project is that it fundamentally changes the way that you reason about your code.

RxJava requires you to think about data as being *pushed* rather than being *pulled* (See Chapter 2: Push vs Pull). While the concept itself is simple, changing a full codebase that is based on a pull paradigm can be a bit daunting. Although consistency is always ideal, you might not always have the privilege to make this transition throughout your entire code base all at once, and so an incremental approach may be needed.

Consider the following:

```java
/**
 * @return a list of users with blogs
 */
public List<User> getUsersWithBlogs() {
    List<User> allUsers = UserCache.getAllUsers();
    List<User> usersWithBlogs = new ArrayList<>();
    for (User user : allUsers) {
        if (user.blog != null && !user.blog.isEmpty()) {
            usersWithBlogs.add(user);
        }
    }
    Collections.sort(
        usersWithBlogs,
        (user1, user2) -> user1.name.compareTo(user2.name)
    );
    return usersWithBlogs;
}
```

This function gets a list of `User` objects from the cache, filters each one based on if the User has a blog or not, sorts them by the `User`'s name, and finally returns them to the caller. Looking at this snippet we notice that much of these operations can take advantage of RxJava operators. Namely, `.filter()` and `.sorted()`.

As the name implies, `.sorted()` is an aggregate operator that sorts emissions from the `Observable`; the emitted object must implement `Comparable`, or alternatively, you can pass a `Comparator` object to specify how the emissions should be sorted.

Rewriting this snippet then gives us:

```java
/**
 * @return a list of users with blogs
 */
public Observable<User> getUsersWithBlogs() {
    return Observable.fromIterable(UserCache.getAllUsers())
        .filter(user -> user.blog != null && !user.blog.isEmpty())
        .sorted((user1, user2) -> user1.name.compareTo(user2.name));
}
```

The first line of the function converts the `List<User>` returned by `UserCache.getAllUsers()` to an `Observable<User>` using the `Observable` creation method `.fromIterable()`. This is the first step into making our code reactive. Now that we are operating on an `Observable`, this enables us to perform any `Observable` operator in the RxJava toolkit: `filter()` and `sorted()` in this case. `.filter()` will filter out users that do not have blogs, and `.sorted()` will sort users that have blogs alphabetically by name.

There are a few other points to note about this change.

First, the method signature is no longer the same. This may not be a huge deal if this method call is only used in a few places and it's easy to propagate the changes up to other areas of the stack; however, if it breaks clients relying on this method, that is problematic and the method signature should be reverted.

Second, RxJava is designed with laziness in mind (See Chapter 2: Lazy Emissions). That is, no long operations should be performed when there are no subscribers to the `Observable`. With this modification that assumption is no longer true since `UserCache.getAllUsers()` is invoked even before there are any subscribers. Whether or not the backing cache is in-memory or on disk, to stay true to RxJava principles, this operation should be postponed until an `Observer` subscribes to it.

Leaving the Reactive world

To address the first issue from our change, we can make use of any of the `.blockingX()` operators available to an `Observable`. Essentially, a `.blockingX()` operator will block the calling thread until an item is emitted downstream. A few notable blocking operators that are available are:

- **.blockingFirst()**: blocks the calling thread and returns the first element emitted by the `Observable`.
- **.blockingNext()**: returns an `Iterable` which allows you to iterate through all the emissions by the `Observable`. Each iteration will block until the `Observable` emits a next item.
- **.blockingLast()**: blocks the calling thread and returns the last element emitted by the `Observable`.
- **.blockingSingle()**: blocks the calling thread until the `Observable` emits an element followed by an `.onComplete()` event, otherwise, an `Exception` is thrown.
- **.blockingSubscribe()**: blocks until a terminal event is received (all elements that are received through `.onNext()` are ignored).

Using a blocking operator to change the method signature back to a `List<User>`, our snippet would now look like:

```java
/**
 * @return a list of users with blogs
 */
public List<User> getUsersWithBlogs() {
    return Observable.fromIterable(UserCache.getAllUsers())
        .filter(user -> user.blog != null && !user.blog.isEmpty())
        .sorted((user1, user2) -> user1.name.compareTo(user2.name))
        .toList()
        .blockingFirst();
}
```

Before calling a blocking operator (i.e. in this case, `.blockingFirst()`) we first need to chain the aggregate operator `.toList()` so that the stream is modified from an `Observable<User>` to a `Single<List<User>>`. Afterwards, we can then call the blocking operator `.blockingFirst()` which unwraps the `Single` and returns a `List<User>`.

A `Single` is a special type of `Observable` that emits a single item. As opposed to the `Observable` which can emit any of the 3 events: `.onNext()`, `onCompleted()`, and `.onError()`, `Single` only has an `.onSuccess()` and an `.onError()`. Using `Single` is preferred when we know that only a single item will be emitted as it offers the caller a clearer intent as to what data will be emitted in the stream (i.e. a single item). We will revisit `Single`s again later in this chapter (See Chapter 5: Completable, Single and Maybe).

Although RxJava supports blocking operations, as much as possible this should be avoided. This is because that instead of being data being pushed as it is ready (the reactive way), we are instead pulling data and blocking the calling thread. In addition, using a blocking operation will cause any exceptions to be thrown on the calling thread; so if the operation were to be invoked on the main thread, the application would crash. When absolutely necessary though, blocking operators are a nice way of stepping out of the reactive world.

The Lazy Approach

As mentioned earlier, RxJava was designed with laziness in mind. That is, long-running operations should be delayed as much as possible until `.subscribe()` is invoked on an `Observable`. To make our solution lazy we can make use of the `Observable` creation methods `.fromCallable()` or `.defer()`.

`.defer()` takes in an `ObservableSource` which is basically a factory that returns an `Observable` whenever the outer `Observable` is subscribed to. In our case, we want to return `Observable.fromItereable(User.getAllUsers())` whenever an `Observer` subscribes.

```
/**
 * @return a list of users with blogs
 */
public Observable<User> getUsersWithBlogs() {
    return Observable.defer(() ->
        Observable.fromIterable(UserCache.getAllUsers())
    ).filter(user -> user.blog != null && !user.blog.isEmpty())
     .sorted((user1, user2) -> user1.name.compareTo(user2.name));
}
```

Now that the long running operation is wrapped in a `.defer()`, we have full control as to what thread this should run on simply by specifying the appropriate `Scheduler` in `.subscribeOn()` and `.observeOn()`.

```
/**
 * @return a list of users with blogs
 */
public Observable<User> getUsersWithBlogs() {
    return Observable.defer(() ->
        Observable.fromIterable(UserCache.getAllUsers())
    ).subscribeOn(Schedulers.io())
     .filter(user -> user.blog != null && !user.blog.isEmpty())
     .sorted((user1, user2) -> user1.name.compareTo(user2.name))
     .observeOn(AndroidSchedulers.mainThread());
}
```

With this change, all subscriptions should now occur on a thread provided by `Schedulers.io()` and the resulting emission will be propagated to the Android main thread. Our code is now fully reactive and subscription should only occur at the moment the data is needed.

Reactive Everything

From the previous examples, we have seen that we can wrap any object in an `Observable` and jump between non-reactive and reactive states using `.blockingX()` operators. In addition, we can delay execution of an operation by wrapping it in a `.fromCallable()` or `.defer()` `Observable` creation method. Using these constructs, we can start converting areas of an Android app to be reactive.

Long Operations

A good place to start using RxJava is whenever you have a process that takes a while to compute—network calls, disk reads and writes, bitmap processing, etc. The following example illustrates a simple function that will write text to the file system.

```java
/**
 * Writes {@code text} to the file system.
 *
 * @param context a Context
 * @param filename the name of the File
 * @param text the String of text to write
 * @return true if the text was successfully written, otherwise, false
 */
public boolean writeTextToFile(
    Context context,
    String filename,
    String text
) {
    FileOutputStream outputStream;
    try {
        outputStream = context.openFileOutput(
            filename, Context.MODE_PRIVATE
        );
        outputStream.write(text.getBytes());
        outputStream.close();
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}
```

When calling this function, we need to make sure that it is done off the main thread since this operation may take a while depending on the text to be written. Imposing such restriction to the caller adds mental load to the developer which increases the likelihood of bugs and it can potentially slow down development. We can prevent such invocations on the main thread by perhaps adding a comment to the function, implying that the method is synchronous in the method name, etc. Although these changes are generally good practice, insuring that the method is called off the main thread is still not guaranteed.

Using RxJava, we can easily wrap this into an `Observable` and specify the `Scheduler` that it should run on. This way the caller doesn't even need to be concerned about invoking the function in a separate thread–the function will guarantee it.

```
/**
* Writes {@code text} to the filesystem.
*
* @param context a Context
* @param filename the name of the File
* @param text the String of text to write
* @return An Observable emitting a boolean indicating
*          whether or not the text was successfully written.
*/
public Observable<Boolean> writeTextToFile(
    Context context,
    String filename,
    String text
) {
    return Observable.fromCallable(() -> {
        FileOutputStream outputStream;
        outputStream = context.openFileOutput(
            filename,
            Context.MODE_PRIVATE
        );
        outputStream.write(text.getBytes());
        outputStream.close();
        return true;
    }).subscribeOn(Schedulers.io());
}
```

Using `.fromCallable()`, writing the text to file is deferred up until subscription time. Since exceptions are first-class objects in RxJava, one other benefit of our change is that we no longer need to wrap the operation in a try/catch, the exception will simply be propagated downstream rather than being swallowed. This allows the caller to handle the exception in a granular way (i.e. show an error to the user depending on what exception was thrown).

One other optimization we can do is return a `Completable` rather than an `Observable`. A `Completable` is a special type of `Observable`–similar to a `Single`–that simply indicates if a computation succeeded via

`.onCompleted()`, or failed via `.onError()`. In this example, returning a `Completable` makes more sense since it seems superfluous to return a single *true* in an `Observable` stream.

```java
/**
* Writes {@code text} to the filesystem.
*
* @param context a Context
* @param filename the name of the File
* @param text the String of text to write
* @return A Completable
*/
public Completable writeTextToFile(
    Context context,
    String filename,
    String text
) {
    return Completable.fromAction(() -> {
        FileOutputStream outputStream;
        outputStream = context.openFileOutput(
            filename,
            Context.MODE_PRIVATE
        );
        outputStream.write(text.getBytes());
        outputStream.close();
    }).subscribeOn(Schedulers.io());
}
```

To complete the operation, we use the `.fromAction()` operation of a `Completable` since the previous return value of *true* is no longer needed.

Completable, Single, and Maybe

So far, the `Observable` class has been our focus for enabling reactive programming with RxJava. However, RxJava also offers the following alternative base reactive types:

Single

In our previous example in Leaving the Reactive World, we have looked at another base reactive type called a `Single`. As mentioned, a `Single` is a special type of `Observable` that only emits a *single* item rather than a stream of items. As a consequence, an observer to a `Single`, a `SingleObserver`, has a different signature and will only receive the following events:

- **.onSubscribe()**: invoked on subscription and a `Disposable` object is passed along which can be used to *unsubscribe* from the `Single`
- **.onSuccess()**: invoked when the `Single` has emitted an item
- **.onError()**: invoked when the `Single` has encountered an error

```java
public interface SingleObserver<T> {

    void onSubscribe(Disposable d);

    void onSuccess(T value);

    void onError(Throwable e);
}
```

Completable

As we've seen in the previous example of writing text to the file system, a `Completable` is another base reactive type that notifies an observer whether or not an operation succeeds. It does not emit an item, it simply tells the observer if the operation passes or fails. An observer to a `Completable`, a `CompletableObserver`, has the following events:

- **.onSubscribe()**: invoked on subscription and a `Disposable` object is passed along which can be used to *unsubscribe* from the `Completable`
- **.onComplete()**: invoked when the `Completable` completes
- **.onError()**: invoked when the `Completable` has encountered an error

```java
public interface CompletableObserver<T> {

    void onSubscribe(Disposable d);

    void onComplete();

    void onError(Throwable e);
}
```

Maybe

A `Maybe` can be thought of as a mix of a `Single` and a `Completable`. It is like a `Single` in that 1 item *may be* emitted, or like a `Completable` in that 0 items *may be* emitted. Its observer, a `MaybeObserver`, has the following events:

- **.onSubscribe()**: invoked on subscription and a `Disposable` object is passed along which can be used to *unsubscribe* from the `Maybe`

- **.onComplete()**: invoked when the `Maybe` completes. This method is invoked when the `Maybe` does not emit an item
- **.onSuccess(T value)**: invoked when the `Maybe` emits a single item
- **.onError()**: invoked when the `Completable` has encountered an error

```
public interface MaybeObserver<T> {

    void onSubscribe(Disposable d);

    void onSuccess(T value);

    void onComplete();

    void onError(Throwable e);
}
```

Since `Single`, `Maybe` and `Completable` offer a subset of capabilities compared to an `Observable`, as you might imagine, the operators that can be performed on them are also limited. For example, the `.toList()` operator is not available on a `Single` or a `.reduce()` operator on a `Completable`.

Collectively, these alternative base reactive types allow you to be more specific in the type system. While it is perfectly valid to use an `Observable` that only emits a single item or an `Observable` that only invokes a terminal event, it is preferred to be as specific as possible as it is more expressive and it informs the caller about the expected behavior of the underlying computation.

If another type is required, however, say when using the operator `.zip()` with a `Single` and with an `Observable`, it is possible to convert between the two types through transformational methods (e.g. `Single`, `Maybe` and `Completable` have the `.toObservable()` method which converts the type to an `Observable`). Conversely, an `Observable` can may be converted to another base reactive type depending on the operator. For example, as we've seen in a previous example invoking `.toList()` on an `Observable` returns a `Single`.

Replacing Callbacks

So far, all the examples that we've looked at emit either one value (i.e. can be modeled as a `Single`), or tell us if an operation succeeded or failed (i.e. can be modeled as a `Completable`).

How might we convert areas in our app that receive continuous updates or events (e.g. location updates, view click events, sensor events, etc.)?

We will look at two ways to do this, first by using `.create()`, and then by using a new construct called `Subject`s.

`.create()` allows us to explicitly invoke any of the available methods of an `Observer` (i.e. `.onNext()`, `.onComplete()` or `.onError()`) as we receive updates from our data source. As we've seen, to use `.create()`, we pass in an `ObservableOnSubscribe` which receives an `ObservableEmitter` whenever an `Observer` subscribes. Using the received emitter, we can then perform all the necessary set-up calls to start receiving updates and then invoke the appropriate emitter event.

In the case of location updates, we can register to receive updates in `.create()` and emit location updates as received.

```
public class LocationManager implements
    GoogleApiClient.ConnectionCallbacks,
    GoogleApiClient.OnConnectionFailedListener {

    GoogleApiClient googleApiClient;

    LocationManager(Context context) {
        this.googleApiClient = new GoogleApiClient.Builder(context)
            .addConnectionCallbacks(this)
            .addOnConnectionFailedListener(this)
            .addApi(LocationServices.API)
            .build();
    }

    @Override public void onConnected(@Nullable Bundle bundle) {
        // googleApiClient connected
    }

    @Override public void onConnectionSuspended(int i) {
        // googleApiClient suspended
    }

    @Override public void onConnectionFailed(
        @NonNull ConnectionResult connectionResult
    ) {
        // googleApiClient failed to connect
    }

    /**
    * Connects the GoogleApiClient.
    */
    void connect() {
        this.googleApiClient.connect();
    }

    /**
    * @return true if GoogleApiClient is connected, otherwise, false
    */
    boolean isConnected() {
        return this.googleApiClient.isConnected();
    }

    /**
    * Call to receive device location updates. `.connect()` must be called f.
    * @return An Observable emitting location updates
    */
    Observable<Location> observeLocation() {
        return Observable.create(emitter -> {

        // Ensure GoogleApiClient is connected
        if (!isConnected()) {
            emitter.onError(
                new Exception("GoogleApiClient not connected")
            );
            return;
        }

        // Create location request
        LocationRequest locationRequest = new LocationRequest();
        locationRequest.setInterval(1000);
        locationRequest.setPriority(
            LocationRequest.PRIORITY_HIGH_ACCURACY
        );

        // Check location permissions
        LocationSettingsRequest.Builder builder =
            new LocationSettingsRequest.Builder()
                .addLocationRequest(locationRequest);
        PendingResult<LocationSettingsResult> pendingResult =
            LocationServices.SettingsApi.checkLocationSettings(
                googleApiClient,
                builder.build()
            );
        pendingResult.setResultCallback(result -> {
            Status status = result.getStatus();

            switch (status.getStatusCode()) {
                case LocationSettingsStatusCodes.SUCCESS:
                    startLocationUpdates(
                        googleApiClient,
                        emitter,
                        locationRequest
                    );
                    break;
                case LocationSettingsStatusCodes.RESOLUTION_REQUIRED:
                    // Resolution required from user
                    break;
                case LocationSettingsStatusCodes.SETTINGS_CHANGE_UNAVAILABLE
                    // Location settings are not satisfied
```

```java
                    break;
                }
            });
        });
    }

    @SuppressWarnings({ "MissingPermission" })
    private void startLocationUpdates(
        GoogleApiClient googleApiClient,
        ObservableEmitter<Location> emitter,
        LocationRequest request
    ) {
        LocationListener listener = location -> {
            if (!emitter.isDisposed()) {
                emitter.onNext(location);
            }
        };

        emitter.setCancellable(() -> {
            LocationServices.FusedLocationApi.
                removeLocationUpdates(
                    googleApiClient,
                    listener
                );
        });
        LocationServices.FusedLocationApi.requestLocationUpdates(
            googleApiClient,
            request,
            listener
        );
    }
}
```

Above, we have created a class `LocationManager` wherein we can start receiving location updates through the method `.observeLocation()`. Once the `GoogleApiClient` is deemed connected, the function inside the `.create()` call first checks if location settings are satisfied, and if so, the function will then request location updates (i.e. the method `.startLocationUpdates()` is invoked). When an updated location is then received, it is emitted to the observer. In addition, we've also added a `Cancellable` to the emitter via `emitter.setCancellable(...)` which ensures that we stop listening to location updates when there are no longer any observers to the created `Observable`.

As we can see here, we essentially replaced the callback-style interface and instead emit the received location in the created `Observable` stream.

Note in the code example above that we are not handling cases when location settings are not satisfied (i.e. cases when `RESOLUTION_REQUIRED` or `SETTINGS_CHANGE_UNAVAILABLE` is received). Handling these cases were left out for brevity but if you want to delve deeper in how to handle these scenarios, consult the Android docs on receiving location updates.

What might happen if multiple observers decide to subscribe to this `Observable`? Since `.create()` returns a *cold* `Observable`, the function inside `.create()` would be repeated on each new subscription which means a new location update request would be made. Clearly this is not what we want and instead we would like to share one subscription but have as many observers as we would like. To accomplish this, we need a way to convert the *cold* `Observable` to a *hot* `Observable`.

Multicasting

The simplest mechanism to convert a given *cold* `Observable` to a *hot* `Observable` is by using the method `.publish()`. Calling `.publish()` returns a `ConnectableObservable` which is a special type of `Observable` wherein each `Observer` shares the same underlying resource. In other words, using `.publish()`, we are able to *multicast* or share to multiple observers.

Subscribing to a `ConnectableObservable`, however, does not begin emitting items.

```
ConnectableObservable<Integer> observable = Observable.create(source -> {
    Log.d(TAG, "Inside subscribe()");
    for (int i = 0; i < 10; i++) {
        source.onNext(i);
    }
    source.onComplete();
}).subscribeOn(Schedulers.io()).publish();

observable.subscribe(i -> {
    // This will never be called
    Log.d(TAG, "Observer 1: " + i);
});

observable.subscribe(i -> {
    // This will never be called
    Log.d(TAG, "Observer 2: " + i);
});
```

For the `Observable` to start emitting items, an explicit call to `.connect()` is required.

```java
ConnectableObservable<Integer> observable = // ...

observable.connect();

// ...observers' onNext() method should now be called
```

`.publish()` is commonly paired with `.refCount()` which implicitly will connect the `Observable` as long as there is at least one subscription (i.e. the `Observable` internally keeps a reference count of observers). The `.publish().refCount()` combo is so common in RxJava that the operator `.share()`—which is basically an alias for `.publish().refCount()`—was introduced.

Rewriting `LocationManager` to support multicasting gives us:

```java
public class LocationManager {

    Observable<Location> locationObservable =
        Observable.create(emitter -> {

            // Ensure GoogleApiClient is connected
            if (!isConnected()) {
                emitter.onError(
                    new Exception("GoogleApiClient not connected")
                );
                return;
            }

            // Create location request
            LocationRequest locationRequest = new LocationRequest();
            locationRequest.setInterval(1000);
            locationRequest.setPriority(
                LocationRequest.PRIORITY_HIGH_ACCURACY
            );

            // Check location permissions
            LocationSettingsRequest.Builder builder =
                new LocationSettingsRequest.Builder()
                    .addLocationRequest(locationRequest);
            PendingResult<LocationSettingsResult> pendingResult =
                LocationServices.SettingsApi.checkLocationSettings(
                    googleApiClient,
                    builder.build()
                );
            pendingResult.setResultCallback(result -> {
                Status status = result.getStatus();

                switch (status.getStatusCode()) {
                    case LocationSettingsStatusCodes.SUCCESS:
                        startLocationUpdates(
                            googleApiClient,
                            emitter,
                            locationRequest
                        );
                        break;
                    default:
                        // Resolution required from user
                        break;
                }
            });
        }).share();

    /**
    * Call to receive device location updates.
    * @return An Observable emitting location updates
    */
    Observable<Location> observeLocation() {
        return locationObservable;
    }
}
```

Simply, we have moved out the `Observable` creation to an instance field of `LocationManager` and have turned it into a multicasted `Observable` via the `.share()` operator.

Subjects

A `Subject` behaves both as an `Observable` and as an `Observer`. `Subject`s are an alternative to multicasting the same resource to multiple subscribers. Implementation-wise, we would want to expose the `Subject` as an `Observable` to clients while expose it, or keep it as a `Subject`, to the provider.

```
public class LocationManager {

   private Subject<Location> locationSubject = PublishSubject.create();

   /**
   * Invoke this method when this LocationManager should
   * start listening to location updates.
   */
   public void connect() {
       this.googleApiClient.connect();
   }

   @Override public void onConnected(@Nullable Bundle bundle) {
        // start listening to location updates
       final LocationRequest locationRequest = new LocationRequest();
       locationRequest.setInterval(1000);
```

```
        locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
        LocationServices.FusedLocationApi.requestLocationUpdates(
            googleApiClient,
            locationRequest,
            location -> {
                locationSubject.onNext(location);
            }
        );
     }

    /**
    * Call to receive device location updates.
    * @return An Observable emitting location updates
    */
    public Observable<Location> observeLocation() {
        return locationSubject.hide();
    }
}
```

In this new implementation, the subtype `PublishSubject` is used which emits events as they arrive starting from the time of subscription. That is, if a subscription is performed at a point when location updates have already been emitted, past emissions will not be received by the observer, only subsequent ones will be emitted. Also notice that the call `.hide()` was invoked on the `Subject` which is generally what you want when exposing the `Subject` externally so that consumers can only use it as an `Observable` and not as an `Observer`.

In addition to `PublishSubject`, RxJava has a few other varieties of `Subject` types:

- **PublishSubject**: emits to an observer only items that are emitted after the time of the subscription
- **AsyncSubject**: emits to an observer only the last value after the `Observable` successfully completes
- **BehaviorSubject**: option subscription, a BehaviorSubject will emit to an observer the most recently emitted item and any subsequent items
- **ReplaySubject**: emits to an observer all previously emitted items and any subsequent items

View Events

UI events such as a `Button` clicks or changes in an `EditText` are another good place to use RxJava. An open source library, RxBinding (developed by

Jake Wharton), provides the necessary bindings to turn `View` events into `Observable` streams.

Using RxBinding, you can turn `View` click events into an `Observable`.

```java
Observable<Object> clicks = RxView.clicks(view);
clicks.subscribe(obj -> {
    // Get click events here
});
```

This example is quite trivial but the benefits of using Rxjava become apparent when used in conjunction with operators. Say for example we wanted to detect double clicks (i.e. 2 clicks that happen within 300 milliseconds apart).

```java
Observable<Object> clicks = RxView.clicks(view);

Observable<List<Object>> clickAggregate =
    clicks.buffer(300, TimeUnit.MILLISECONDS);

Observable<List<Object>> doubleClicks =
    clickAggregate.filter(list -> list.size() >= 2);

doubleClicks.subscribe(o -> {
    Timber.d("Received double clicks.");
});
```

Normally the above method calls to the click `Observable` would be chained, but for educational purposes I've separated it out line-by-line so we can understand the transformations at each step.

First, we convert click events from `view` to an `Observable` using `RxView.clicks()`. After that, we use the operator `.buffer()` which buffers items that occur within a provided timespan–in this case 300 milliseconds–into a `List<Object>`. Afterwards, we simple filter each emission of the `.buffer()` operation using `.filter()` to check if the buffered list contains 2 or more items, if it does, then a double click was just performed.

We will look into the `.buffer()` operator more in Chapter 6: Buffering.

Disposable and the Activity/Fragment Lifecycle

So far we have overlooked a key component in a reactive stream–the `Disposable`. A `Disposable` is returned as a result from a subscription so that we can control when to unsubscribe so that the underlying `Observable` can stop emitting events. Failure to do so can cause unwanted memory leaks that are hard to trace.

When dealing with the Activity/Fragment lifecycle, a good place to unsubscribe would be when the Activity or Fragment is no longer needed. That is, in the `.onDestroy()` lifecycle.

```java
public class MyActivity extends Activity {
     Disposable disposable;

     protected void onCreate(Bundle savedInstanceState) {
         this.disposable = // ...subscribe to some Observable
     }

     protected void onDestroy() {
         if (!disposable.isDisposed()) {
             disposable.dispose();
         }
     }
}
```

RxJava also has classes that implement `DisposableContainer`, such as `CompositeDisposable`, to make disposing from multiple `Disposable` instances easy.

```java
public class MyActivity extends Activity {
     CompositeDiposable disposables = new CompositeDisposable();

     protected void onCreate(Bundle savedInstanceState) {
         Disposable disposable1 = // ...subscribe to some Observable
         Disposable disposable2 = // ...subscribe to another Observable
         disposables.addAll(disposable1, disposable2);
     }

     protected void onDestroy() {
         if (!disposables.isDisposed()) {
             disposables.dispose();
         }
     }
}
```

In general, unsubscribing from `Disposable`s in an Activity/Fragment `.onDestroy()` method is a good guideline however, ideally unsubscribing should be done as soon as a resource is no longer needed. For example, say an `Observable`'s events is only needed when an `Activity` is in the foreground. In this case, subscription should occur on the `Activity`'s `.onResume()` method and unsubscription should occur in the opposing lifecycle method–`.onPause()`.

RxLifecycle

Alternatively, an open source library called RxLifecycle, created by Trello, can be used to automatically unsubscribe from an `Observable` once an Activity/Fragment lifecycle event occurs. This way, you don't have to manage the `Disposable` object explicitly, RxLifecycle will handle it for you.

To bind a `Disposable` with RxLifecycle, you can specify a lifecycle event when an `Observable` should be unsubscribed (e.g. on an Activity's `.onDestroy()`):

```
observable
    .compose(RxLifecycle.bindUntilEvent(
        lifecycle,
        ActivityEvent.DESTROY
    ))
    .subscribe();
```

Or you can let RxLifecycle decide when it should unsubscribe from an `Observable`. In this case, it will unsubscribe at the opposing lifecycle event from when the `Observable` was subscribed (i.e. `.onCreate()`/`.onDestroy()`, `.onStart()`/`.onStop()` or `.onResume()`/`.onPause()`).

```
observable
    .compose(
        RxLifecycleAndroid.bindActivity(lifecycle)
    )
    .subscribe();
```

In both examples, **lifecycle** represents an `Observable<ActivityEvent>` which can be obtained by subclassing `RxActivity`/`RxFragment`:

```
public class MyActivity extends RxActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        observable
            .compose(RxLifecycle.bindUntilEvent(
                lifecycle,
                ActivityEvent.DESTROY
            ))
            .subscribe();
    }
}
```

Or alternatively, you can also create an `Observable<ActivityEvent>` yourself:

```
public class MyActivity extends Activity {
    private BehaviorSubject<ActivityEvent> lifecycleSubject =
        BehaviorSubject.create();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        lifecycleSubject.onNext(ActivityEvent.CREATE);
        observable
            .compose(RxLifecycle.bindUntilEvent(
                lifecycle,
                ActivityEvent.DESTROY
            ))
            .subscribe();
    }

    @Override
    protected void onStart() {
        super.onStart();
        lifecycleSubject.onNext(ActivityEvent.START);
    }

    @Override
    protected void onResume() {
        super.onResume();
        lifecycleSubject.onNext(ActivityEvent.RESUME);
    }

    @Override
    protected void onPause() {
        lifecycleSubject.onNext(ActivityEvent.PAUSE);
        super.onPause();
    }

    @Override
    protected void onStop() {
        lifecycleSubject.onNext(ActivityEvent.STOP);
        super.onStop();
    }

    @Override
    protected void onDestroy() {
        lifecycleSubject.onNext(ActivityEvent.DESTROY);
        super.onDestroy();
    }
}
```

In addition, RxLifecycle can also be used in conjunction with the new `Lifecycle` class in the Android Architecture Components library:

```
public class MyActivity extends LifecycleActivity {
    private LifecycleProvider<Lifecycle.Event> provider =
        AndroidLifecycle.createLifecycleProvider(this);

    @Override
    protected void onCreate() {
        super.onCreate();
        myObservable
            .compose(provider.bindToLifecycle())
            .subscribe();
    }
}
```

Chapter 6: Backpressure

By now, you should be familiar with the distinction of *push vs pull* in reactive interfaces. Instead of the traditional model of requesting data from a source (i.e. pulling data), the source will push data to an observer as the data is ready (See Chapter 2: Push vs Pull). One problem we overlooked though is: what happens if a consumer can't keep up with a producer? That is, what happens if a producer (`Observable`) produces data faster than a consumer (`Observer` or an `Operator`) can consume? As you might expect, this situation puts pressure on the system and can cause bugs if handled incorrectly. In this chapter, we will dive deeper into this problem and look into different ways to solve it.

Fast Producer, Slow Consumer

Say given a list of image files, we were to load each image file to memory and apply a CPU-bound function `.processBitmap()` which takes some time to compute.

```
File[] imageFiles = // image files to process

Observable<File> fileStream = Observable.fromArray(
    imageFiles
).subscribeOn(Schedulers.computation());

Observable<Bitmap> bitmapStream =
    fileStream.map(file -> BitmapFactory.decodeFile(file.getAbsolutePath()));

Observable<Pair<Bitmap, Integer>> bitmapZippedStream = Observable.zip(
    bitmapStream,
    Observable.range(1, imageFiles.length),
    Pair::new
).doOnNext(pair -> Log.d(TAG, "Produced bitmap: " + pair.second));

bitmapZippedStream.observeOn(
    AndroidSchedulers.mainThread()
).subscribe(pair -> {
    processBitmap(pair.first);
    Log.d(TAG, "Processed bitmap: " + pair.second);
});
```

```java
private void processBitmap(Bitmap bitmap) {
    // Simulate long operation
    try {
        new Thread(() -> {
        }).sleep(300);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return bitmap;
}
```

The above code should be fairly straightforward: first, a list of image files are converted into an `Observable<File>`. Using `.map()`, each emission from that stream is then decoded into a bitmap which produces an `Observable<Bitmap>`. The bitmap stream is then zipped with an integer stream (i.e. `Observable.range(1, imageFiles.length)`) and combined as a `Pair`. Each decoded bitmap is then printed on Logcat. Finally, in the `.subscribe()` function, each bitmap is processed via a long operation `.processBitmap()` (a thread sleep of 300 milliseconds is performed to simulate a this).

Can you guess what running the above code might produce?

Given a sufficiently long list of large images (1-5 Mb), the above code may throw an `OutOfMemoryError` (in RxJava 1.x, the above code would have produced a `MissingBackpressureException`, however, this was changed since RxJava 2.x). The reason for this is because the producer will continue to emit bitmaps despite the consumer not being able to keep up with the emissions. In turn, the emitting `Observable` maintains an internal unbounded buffer of the generated bitmaps and if the consumer cannot keep up, the buffer will continue to grow until the program runs out of memory. To solve this, we need a way to notify upstream that it should slow down its production until the consumer downstream can keep up with processing items.

Backpressure

The mechanism by which we can notify upstream that it should slow down its production is called *backpressure*. Using an `Observable` however, we cannot apply backpressure as `Observable`s are not designed to support backpressure. For this, we will need another base reactive class called `Flowable`.

Flowable

In a word, `Flowable` is a base reactive class that is backpressure-enabled. If you know you are dealing with a stream that you might want to add backpressure to, a `Flowable` is what you want. `Flowable` supports the following backpressure strategies:

- **BackpressureStrategy.ERROR** - produces a `MissingBackpressureException` in the case when downstream cannot keep up with item emissions
- **BackpressureStrategy.BUFFER** - buffers all items until downstream consumes it (default buffer size is 128)
- **BackpressureStrategy.DROP** - drops the most recent item if downstream cannot keep up
- **BackpressureStrategy.LATEST** - keeps only the latest item overwriting any previous value if downstream cannot keep up
- **BackpressureStrategy.MISSING** - no backpressure is added to the `Flowable` (this is equivalent to using an Observable)

So when might you want to choose a `Flowable` over an `Observable`? The RxJava wiki, offers the following general guidelines:

When to use Observable

- You have a flow of no more than 1000 elements at its longest: i.e., you have so few elements over time that there is practically no chance for OOME (OutOfMemoryError) in your application.
- You deal with GUI events such as mouse moves or touch events: these can rarely be backpressured reasonably and aren't that frequent. You may be able to handle an element frequency of 1000 Hz or less with Observable but consider using sampling/debouncing anyway.
- Your flow is essentially synchronous but your platform doesn't support Java Streams or you miss features from it. Using Observable has lower overhead in general than Flowable. (You could also consider IxJava which is optimized for Iterable flows supporting Java 6+).

When to use Flowable

- Dealing with 10k+ of elements that are generated in some fashion somewhere and thus the chain can tell the source to limit the amount it generates.
- Reading (parsing) files from disk is inherently blocking and pull-based which works well with backpressure as you control, for example, how many lines you read from this for a specified request amount).
- Reading from a database through JDBC is also blocking and pull-based and is controlled by you by calling ResultSet.next() for likely each downstream request.
- Network (Streaming) IO where either the network helps or the protocol used supports requesting some logical amount.
- Many blocking and/or pull-based data sources which may eventually get a non-blocking reactive API/driver in the future.

Rewriting our example using backpressure with `Flowable` gives us:

```java
File[] imageFiles = // image files to process

Flowable<File> fileStream = Flowable.fromArray(
    imageFiles
).subscribeOn(Schedulers.computation());

Flowable<Bitmap> bitmapStream =
    fileStream.map(file -> BitmapFactory.decodeFile(file.getAbsolutePath()));

Flowable<Pair<Bitmap, Integer>> bitmapZippedStream = Flowable.zip(
    bitmapStream,
    Flowable.range(1, imageFiles.length),
    Pair::new
).doOnNext(pair -> Log.d(TAG, "Produced bitmap: " + pair.second));

bitmapZippedStream.observeOn(
    AndroidSchedulers.mainThread()
).subscribe(pair -> {
    processBitmap(pair.first);
    Log.d(TAG, "Processed bitmap: " + pair.second);
});
```

With this change, we replaced all references of `Observable` to `Flowable`. By default, the backpressure strategy used by `Flowable` is to impose a *bounded* buffer–the buffer is filled by the producer and production will halt until most items in the buffer are consumed by the consumer. It may still be the case, however, that the above code would produce an `OutOfMemoryError`. This is because the default buffer size is set to be 128 items and depending on your use-case, that size might be too much. To mitigate this, we can specify a different buffer capacity in an overloaded version of `.observeOn()`

```
// ...
int bufferSize = 10;
bitmapZippedStream.observeOn(
    AndroidSchedulers.mainThread(),
    false,
    bufferSize
).subscribe(pair -> {
    // ...
});
```

Here we have changed the bounded buffer size to only contain 10 items. The second parameter to `.observeOn()` indicates whether or not an `.onError()` notification should cut ahead of `.onNext()` notifications, in this case, specifying `false` indicates that it can. Running the above code with backpressure should print out:

```
Produced bitmap: 1
Produced bitmap: 2
...
Produced bitmap: 10
Processed bitmap: 1
Processed bitmap: 2
...
Processed bitmap: 7
Processed bitmap: 8
Produced bitmap: 11
...
```

As you can see, production of bitmaps halted once the set buffer size of 10 has been reached. The consumer was then allowed to catch up and production continued when the buffer was close to empty (in this case, production continued once the buffer size contained 2 items).

Backpressure using Subscriber

Fine-grained control of how a consumer notifies a producer that it can consume more data can be done by using a `Subscriber`. In addition to the standard observer methods `.onNext(T)`, `.onError(Throwable)`, and `.onComplete()`, a `Subscriber` has an additional method, `.onSubscribe(Subscription)`, which is called upon subscribing to a producer (i.e. `Flowable`). The `Subscription` object received from `.onSubscribe(Subscription)` is then used to control the flow of production from the producer through its method `.request(long)`. As soon as `.onSubscribe()` is invoked, we must immediately request data from the producer via `.request(long)` to begin receiving items.

```java
int bufferSize = 10;
bitmapZippedStream.observeOn(
    AndroidSchedulers.mainThread(),
    false,
    bufferSize
).subscribe(new Subscriber<Pair<Bitmap, Integer>>() {

    Subscription subscription;

    @Override
    public void onSubscribe(Subscription s) {
        subscription = s;
        subscription.request(1);
    }

    @Override
    public void onNext(Pair<Bitmap, Integer> pair) {
        // Do something with `pair`
        subscription.request(1);
    }

    @Override
    public void onError(Throwable throwable) {
    }

    @Override
    public void onComplete() {
    }
});
```

Above, we request 1 item to be produced as soon as the `Subscriber` is subscribed. Afterwards, for each `.onNext()` item, we request 1 more item to be produced. There's nothing very special here and it essentially behaves the same way as before. If however our consumption of data should be limited and we want to communicate this to the producer (e.g. say we were caching the bitmaps and we want to stop receiving them when the cache is full) we can do that by conditionally requesting from the `Subscription` object in `.onNext()`.

```
.subscribe(new Subscriber<Pair<Bitmap, Integer>>() {
// ...
    @Override
    public void onNext(Pair<Bitmap, Integer> pair) {
        // Do something with `pair`
        if (shouldReceiveMoreBitmaps()) {
            subscription.request(1);
        }
    }
// ...
});
```

Throttling and Buffering Items

In addition to backpressure, there are several handy operators that can be used to solve the problem of a fast producer but a slow consumer. These operators allow you to either *throttle* or *buffer* items so that a consumer can keep up. Oftentimes, these operators are preferred over backpressure as they are much simpler to work with.

Throttling

Let's look at an Android-specific example where we might want to apply these principles–a device's accelerometer events. First, let's create a class called `DeviceSensorManager` that can be queried for an `Observable` that emits accelerometer events.

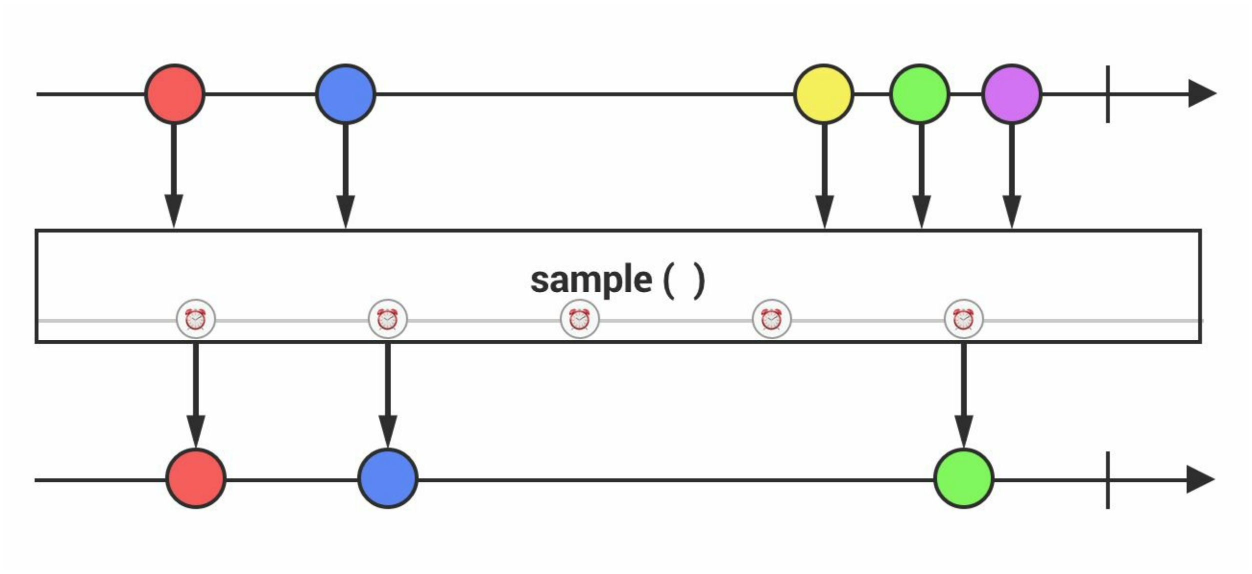
```
public class DeviceSensorManager {

    private Observable<SensorEvent> accelerometerEventObservable;

    public DeviceSensorManager(Context context) {
        accelerometerEventObservable =
            Observable.create(emitter -> {
                SensorEventListener sensorEventListener =
                    new SensorEventListener() {

                    @Override public void onSensorChanged(SensorEvent e) {
                        if (e.sensor.getType() == Sensor.TYPE_ACCELEROMETER)
                            emitter.onNext(e);
                        }
                    }

                    @Override public void onAccuracyChanged(
                        Sensor sensor,
                        int accuracy
                    ) { }
                };

                SensorManager sensorManager =
                    (SensorManager) context.getSystemService(
                        Context.SENSOR_SERVICE
                    );
                sensorManager.registerListener(
                    sensorEventListener,
                    sensorManager.getDefaultSensor(
                        Sensor.TYPE_ACCELEROMETER
                    ),
                    SensorManager.SENSOR_DELAY_NORMAL
                );

                // Clean up when there are no longer any subscribers
                emitter.setCancellable(() -> {
                    sensorManager.unregisterListener(
                        sensorEventListener
                    );
                });
            }
        }).share();
    }

    public Observable<SensorEvent> accelerometerEventObservable() {
        return accelerometerEventObservable;
    }
}
```

When creating a `DeviceSensorManager`, we pass it a `Context` so that we can obtain a handle to a `SensorManager`. Given the `SensorManager`, we then set a `SensorEventListener` that listens for accelerometer events (i.e. events of type `Sensor.TYPE_ACCELEROMETER`) at a rate defined by `SensorManager.SENSOR_DELAY_NORMAL`. This listener is then wrapped in an `Observable` and all accelerometer events are emitted down the `Observable` stream. Lastly, the `Observable` is multicasted using `.share()` so that multiple `SensorEventListener`s are not registered on each subscription. The listener is also unregistered from the `SensorManager` when there are no longer any subscribers (i.e. the lambda inside `emitter.setCancellable(...)` will be invoked when all observers of the `Observable` are unsubscribed/disposed).

The rate at which accelerometer events are emitted is defined by `SensorManager.SENSOR_DELAY_NORMAL` (i.e. ~20 milliseconds). But say for our needs we wanted to consume events at a much slower rate; how might we do that? Surely, we can attach a separate listener with the desired rate but what if we want to use the same `Observable`? There are a couple of options to achieve this, one of which is to use the filtering operator `.sample()`.

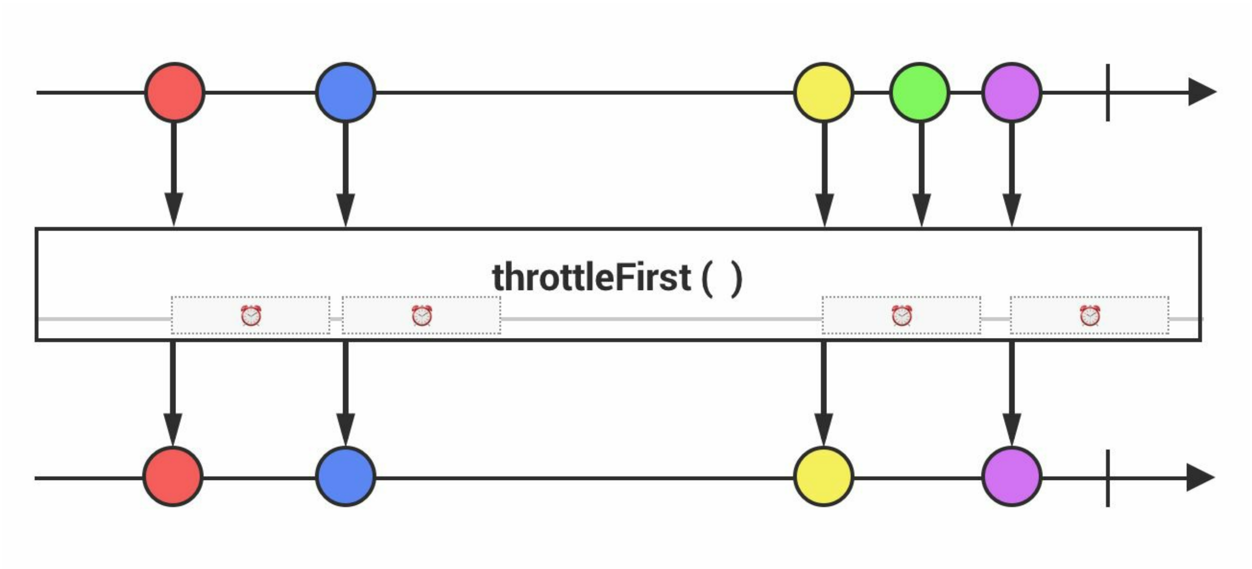
Sample & Throttle

`.sample(long, TimeUnit)` works by periodically sampling events at a specified rate by emitting the latest item in a period while dropping all other items.

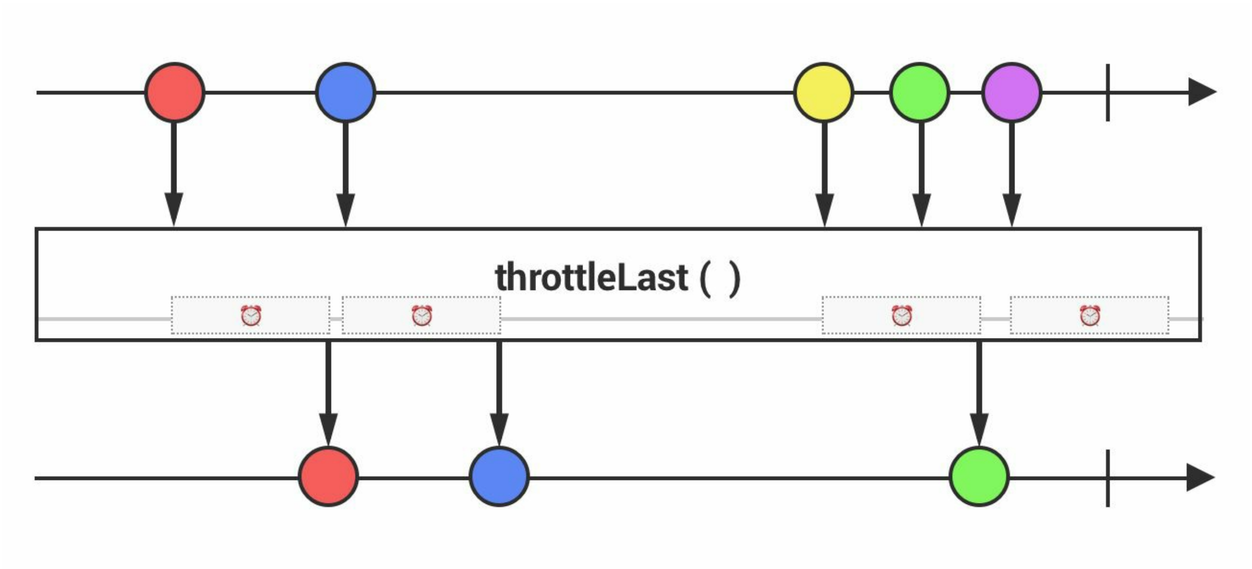


Marble diagram of .sample()

`.throttleFirst()` and `.throttleLast()` on the other hand can be used to only emit the first or last item, respectively, over a sequential time period.



Marble diagram of .throttleFirst()



Marble diagram of .throttleLast()

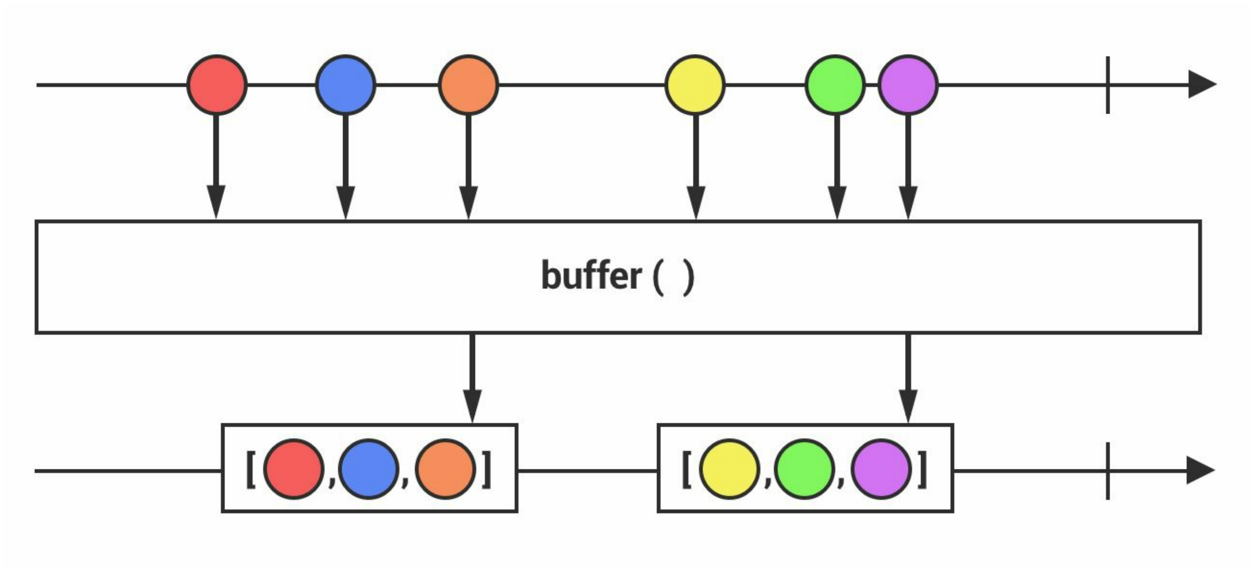
Buffering

Another strategy to help mitigate problems with chatty producers is to buffer or collect emissions and emit the result downstream at a less frequent rate. Two useful operators for achieving this are `.buffer()` and `.window()`.

Buffer & Window

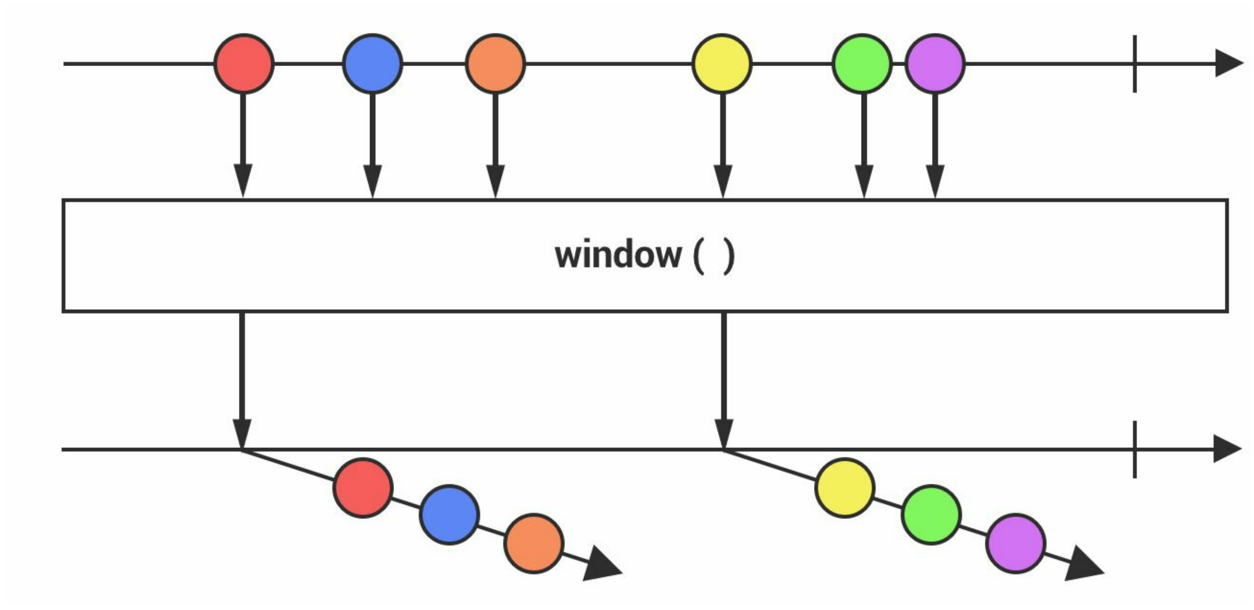
As the name implies, `.buffer(...)` allows you to buffer emissions from upstream into a list. `.buffer()` is overloaded to support buffering given different criteria, a few handy ones are:

- **Observable.buffer(int count)**: buffers items *count* items
- **Observable.buffer(ObservableSource<Type> boundary)**: buffers items in between emissions from *boundary*
- **Observable.buffer(long timespan, TimeUnit timeUnit)**: buffers items that are emitted in a given timespan



Marble diagram of .buffer()

A similar operator to `.buffer()`, `.window()`, allows you to emit `Observable`s instead of lists. `.window()` is also overloaded and allows buffers given different criteria.



Marble diagram of .window()

Why might you want a stream of `Observable`s instead of lists? It's helpful to think of this difference in terms of the difference between `.map()` and `.flatMap()` as we've seen in Chapter 3 (See Chapter 3: .flatMap()). With a stream of `Observable`s, we can further chain operators and even add concurrency (See Chapter 4: Concurrency with .flatMap()).

Chapter 7: Error Handling

In Java, when an error occurs in the normal flow of a program, an `Exception` is thrown. A try/catch statement can then be used to recover from the Exception and perform the appropriate fallback behavior (i.e. ignore the error and continue, retry the operation, display an error message to the user etc.). Most Exceptions that are handled in this manner are *checked* exceptions, that is, the thrown Exception needs to be declared and explicitly handled for our code to compile. *Unchecked* exceptions however, are harder to catch and if not handled, can cause our application to crash. Oftentimes, this is not the behavior that we want since we want apps to be perform well despite these edge cases.

In RxJava, errors are treated slightly different. Rather than being seen as *exceptions* from the normal flow of a program, errors are considered to be a part of the normal flow of the program and thus should be accounted for when dealing with a given stream. In this chapter, we will look at different ways of handling errors in an `Observable` to help create resilient applications.

Errors Along the Chain

As we've seen earlier, one of the events an `Observer` can receive is the `.onError(Throwable)` event. `.onError()` is considered a terminal such that once triggered, the `Observable` will no longer emit any other event. There are several ways to invoke the `.onError()` notification, some are explicit such as invoking the `.onError()` on an `Emitter`:

```
Observable<Integer> observable = Observable.create(emitter -> {
    emitter.onError(new Exception("An error occurred."));
});
observable.subscribe(i -> {
}, throwable -> {
    Log.e(TAG, "onError(): " + throwable.getMessage());
});
```

Or by returning an `Observable.error()` wherever an `Observable` is expected such as in the `.flatMap()` operator:

```
Observable<Integer> observable =
    Observable.range(0, 10).flatMap(i -> {
        return (i == 3)
            ? Observable.error(new Exception("An error occurred"))
            : Observable.just(i);
    });
observable.subscribe(i -> {
}, throwable -> {
    Log.e(TAG, "onError(): " + throwable.getMessage());
});
```

In addition to these approaches, `Exception`s are also propagated to the observer whenever they are thrown in a function contained within the `Observable` sequence. As one might notice, all methods in interfaces in the **io.reactivex.functions.*** are defined to throw an `Exception`. The reason being is that these functions are invoked in a try/catch statement internally so that when an error is encountered, that error is propagated to the observer.

```
Observable<Integer> observable =
    Observable.range(0, 10).map(i -> {
        if (i == 3) {
            throw new Exception("An error occurred.");
        }
        return i * 2;
    });
observable.subscribe(i -> {
}, throwable -> {
    Log.e("onError(): " + throwable.getMessage());
});
```

There are some Exceptions, however, that are too fatal to recover from and are not propagated down to the observer (i.e. instances of the following classes: `VirtualMachineError`, `ThreadDeath`, and `LinkageError`). Instead, these Exceptions are thrown from the thread where the Exception occurred.

```
Observable<Integer> observable =
    Observable.range(0, 10).subscribeOn(
        Schedulers.io()
    ).map(i -> {
        if (i == 3) {
           throw new OutOfMemoryError();
        }
        return i * 2;
    });

```

```
observable.observeOn(
    Schedulers.computation()
).subscribe(i -> {
}, throwable -> {
    Log.e("onError(): " + throwable.getMessage());
});
```

In the above code, the `OutOfMemoryError` won't be captured in the `.onError()` call since OOMs are considered fatal. Instead, the Exception will be thrown on a thread from `Schedulers.io()`.

Delivering Errors

As we've seen, when an error is encountered in a stream, RxJava will deliver the error to the `Observer` via `.onError()`. Given that `.onError()` is terminal, no further processing will occur. This however can be problematic as Dan Lew explains in his blog post "Error Handling in RxJava":

> I want to clear up something that many RxJava beginners get wrong: onError is an extreme event that should be reserved for times when sequences cannot continue. It means that there was a problem in processing the current item such that no future processing of any items can occur.
>
> It's the Rx version of try-catch: it skips all code between the exception and onError. Sometimes you want this behavior: suppose you're downloading data then saving it to the disk. If the download fails, you want to skip saving. But what about if you're, say, polling data from a source? If one poll fails you still want to continue polling. Or maybe, in the previous example, you want to save dummy data in the case that the download fails.
>
> In other words: often times, instead of calling onError and skipping code, you actually want to call onNext with an error state. It's much easier to handle problems in onNext since you still get to run all your code and the stream isn't terminated.

Indeed, depending on what we are processing, we might want to handle an error differently. In the next section, we will look at the different error-handling related operators that are available in RxJava.

Error-Handling Operators

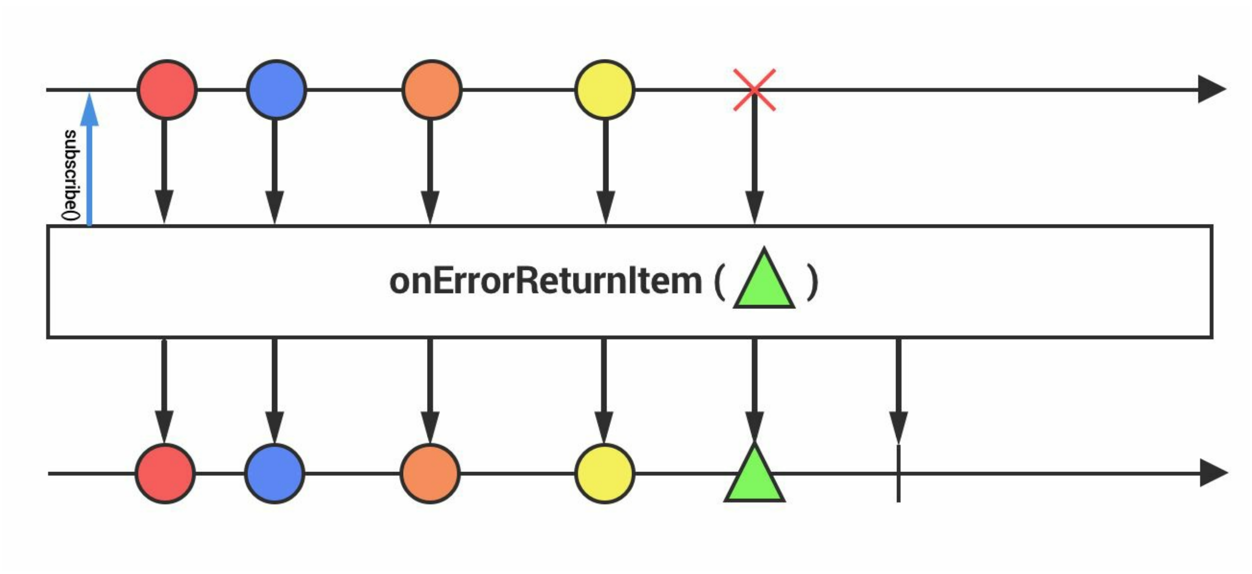
Say we were to implement an image store such that when a request comes in it will first check if the image is cached, and if so, return the image from the cache, otherwise, return fetch image over the network.

```java
public class ImageStore {

    private ImageStoreCache cache = new ImageStoreCache();

    public Observable<Bitmap> getImage(String filename) {
        return cache.getImage(filename);
    }
}

public class ImageStoreCache {

    private LruCache<String, Bitmap> cache = new LruCache<>(100);

    public Observable<Bitmap> getImage(String filename) {
        return Observable.fromCallable(() -> {
            Bitmap cachedBitmap = cache.get(filename);
            if (cachedBitmap != null) {
                return cachedBitmap;
            }
            throw new ImageNotInCacheException();
        });
    }


    public void addImage(String filename, Bitmap bitmap) {
        cache.put(filename, bitmap);
    }
}

public class ImageNotInCacheException extends Exception {
}
```

`ImageStore` so far will simply request the image from the `ImageStoreCache` and will not yet make a network call if an image does not exist in the cache. Instead, it will return the `ImageNotInCacheException` to the Observer’s `.onError()` method. To handle the case of fetching an image over the network, we can make use of any of the following error-handling operators:

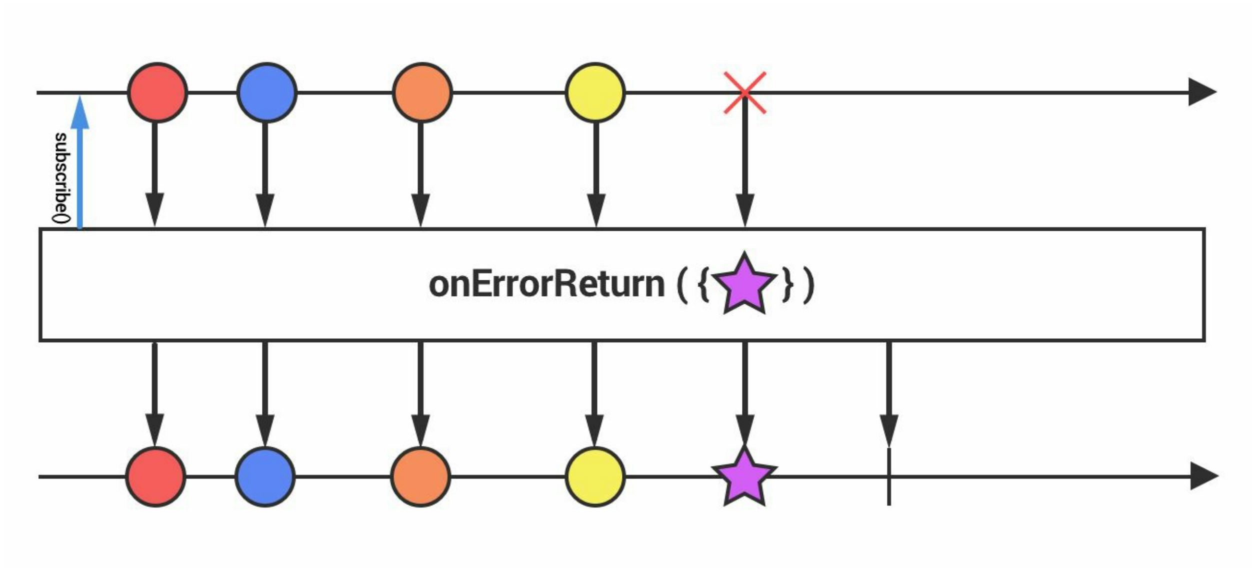
Observable.onErrorReturnItem(), Observable.onErrorReturn() and Observable.onErrorResumeNext()

- **Observable.onErrorReturnItem(T item)**: when an error is received upstream, this operator will return the item provided to it downstream instead of the received error.



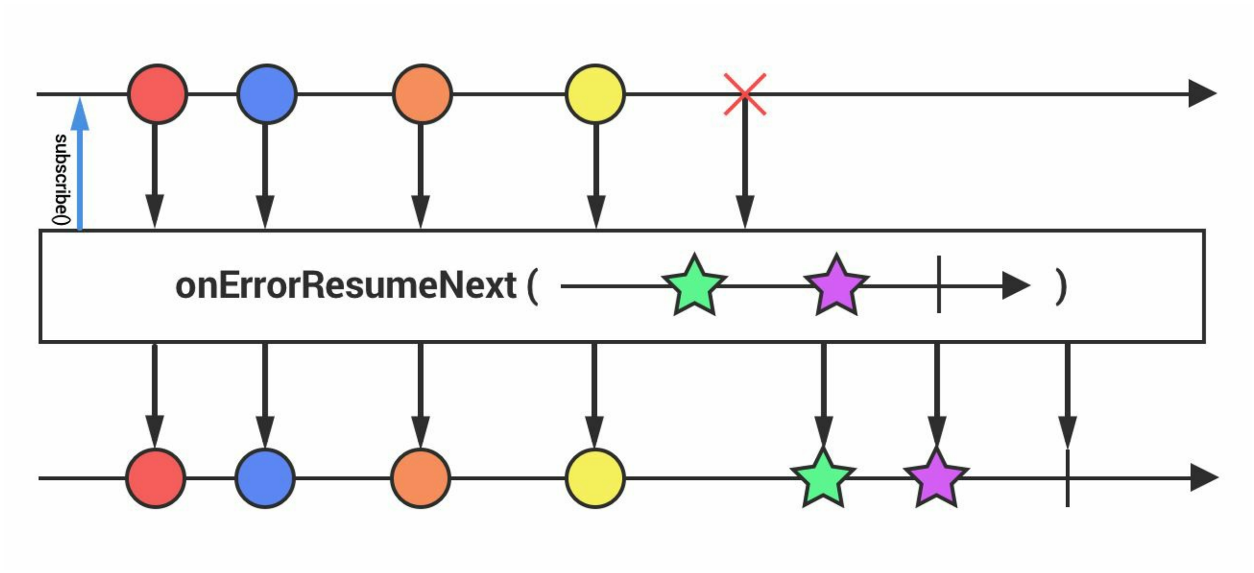
Marble diagram of .onErrorReturnItem()

- **Observable.onErrorReturn(Function<? super Throwable, ? extends T> valueSupplier)**: when an error is received upstream, this operator will invoke the function provided and return the result of that function downstream instead of the received error. This operator is preferred if you want to return a different item depending on the error.



Marble diagram of .onErrorReturn()

- **Observable.onErrorResumeNext(ObservableSource<? extends T> next))**: similar to `.onErrorReturnItem()` but instead of returning an item, it will pass control to the provided `Observable` instead of passing the received error.



Marble diagram of .onErrorResumeNext()

In our case, we will make use of `.onErrorResumeNext()` to return an `Observable` when an `ImageNotInCacheException` error is received.

```java
public class NetworkClient {
    Observable<Bitmap> fetchImage(String filename) {
        // Network call to fetch image here
    }
}

public class ImageStore {

    private NetworkClient networkClient;
    private ImageStoreCache cache = new ImageStoreCache();

    public ImageStore(NetworkClient networkClient) {
        this.networkClient = networkClient;
    }

    public Observable<Bitmap> getImage(String filename) {
        return cache.getImage(filename)
                    .onErrorResumeNext(
                        networkClient.fetchImage(
                            filename
                        ).doOnNext(bitmap -> cache.addImage(filename, bitmap)
                    );
    }
}
```

When an error is received from the cache, we retrieve an image over the network via `.fetchImage()`. Finally, once the image is retrieved over the network, the cache is then updated to store the retrieved bitmap.

Retry Operators

There are some operations that if they fail to complete require retrying either immediately or at a later point in time. Say for example we are developing a chat application and a user sends a message to another user. If sending that message fails because of a bad network connection, that message should get resent at a better point in time (e.g. when the sender establishes a better network connection). While there are many ways to solve this problem, a simple solution would be to add retry logic until the operation succeeds.

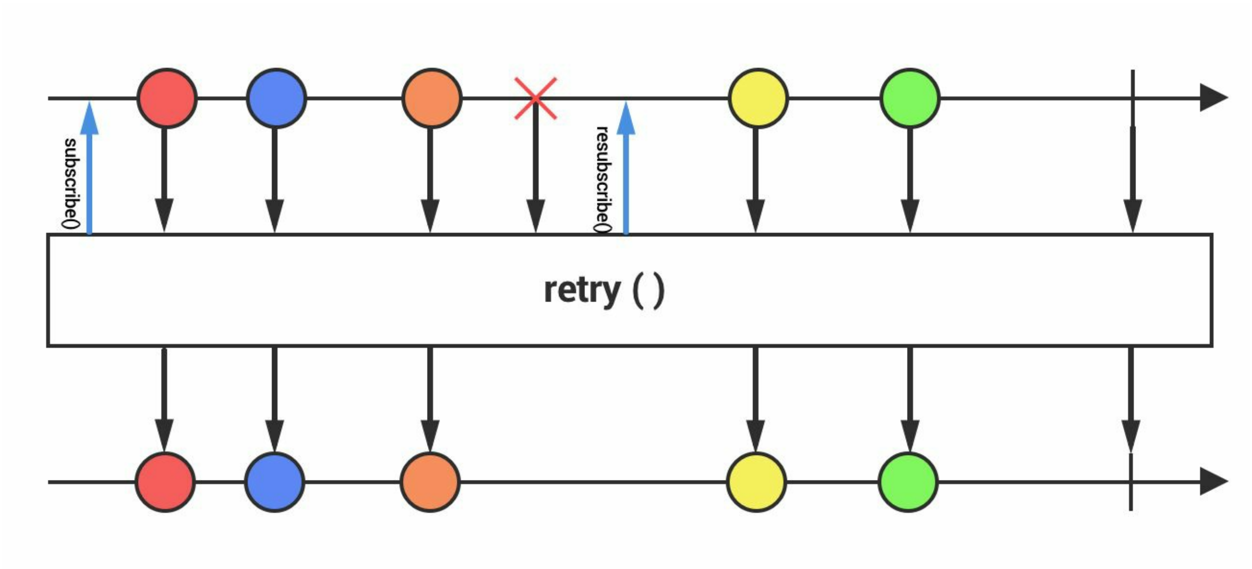
```java
public class Message {
    private String sender;
    private String receiver;
    private String text;

    public Message(String sender, String receiver, String text) {
        this.sender = sender;
        this.receiver = receiver;
        this.text = text;
    }
}

public class NetworkClient {

    Observable<Message> sendMessage(Message message) {
        // Send message network call here
    }
}
```

Above, we have a simple `Message` POJO that contains some text as well as the sender and the receiver of the message. In addition, we've added a method in `NetworkClient` to send a `Message` to the receiver which returns an `Observable<Message>` that emits the sent message to the observer if the network call succeeds, otherwise, it emits an error.

```
Message message = new Message("chris, "angus", "Yo");
networkClient.sendMessage(message).subscribe(() -> {
    // Message sending succeeds
}, throwable -> {
    // Message sending fails
});
```

Observable.retry()

A straightforward approach to retry a failed operation is to use `.retry()`. `.retry()` will intercept error notifications received upstream and not pass those through its observers. Instead, it will resubscribe to the source `Observable` and give it the opportunity to complete its sequence without error.



Marble diagram of .retry()

```
Message message = new Message("chris, "angus", "Yo");
networkClient.sendMessage(message).retry(10).subscribe(() -> {
    // Message sending succeeds
}, throwable -> {
    // Message sending fails
});
```

Above, we have modified sending a message to retry at most 10 times until it succeeds.

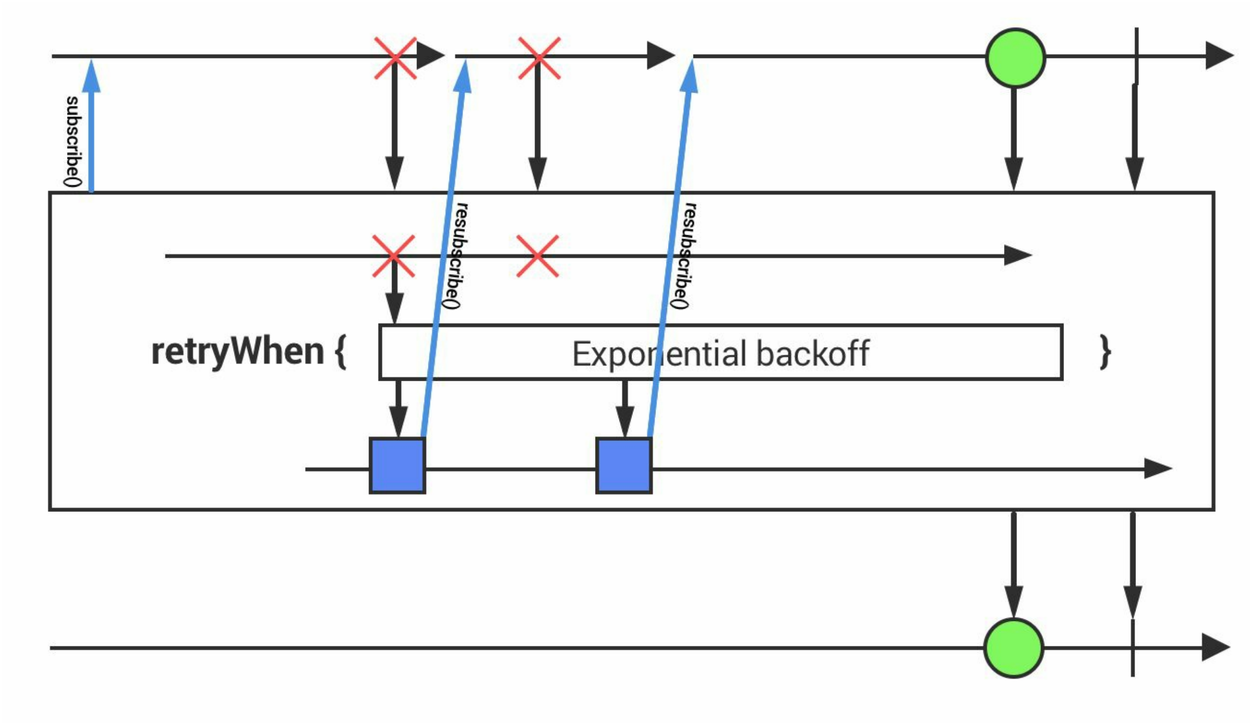
It might also be useful to inspect the error before blindly retrying the operation. For example, say the error returned is because the user that is trying to send a message is no longer authenticated. In this case, we do not want to retry the operation but instead report the error to the observer. To do this, we can use an overloaded version of `.retry()` which allows us to pass a predicate to inspect the error to retry conditionally.

```
networkClient.sendMessage(message).retry(10, error -> {
    return !(error instanceof AuthenticationError);
}).subscribe(() -> {
    // Message sending succeeds
}, throwable -> {
    // Message sending fails
});
```

While retrying an operation immediately after failure a number of times may be sufficient in some cases, in others cases such as network issues, retrying immediately can exacerbate the issue and a different approach for retrying is necessary.

Observable.retryWhen(Function<? super Observable<Throwable>, ? extends ObservableSource<?>> handler)

One common way of retrying when it comes to networking issues is to use *exponential backoff* with the rate of retries. That is, the delay of a subsequent retry would be some multiple of the delay of the previous one. To do this, we can make use of `.retryWhen()`.



Marble diagram of .retryWhen()

Using `.retryWhen()`, we provide it a function handler that receives errors in the form of an `Observable<Throwable>` and the handler in turn would return an `Observable<?>`. The emission value of the returned `Observable` is ignored–hence the wildcard–the operator only cares about the `.onNext()` notification event which signals when the retry should be performed.

```java
Observable<Message> sendMessageObservable = networkClient.sendMessage(message

sendMessageObservable.retryWhen(throwable -> {
    Observable<Long> retrySignal =
        throwable.zipWith(Observable.range(0, 6), (t, i) -> i).flatMap(i -> {
        final long delay = (long) Math.pow(2, i);
        Log.d(TAG, "Retry delayed by: " + delay);
        return Observable.timer(delay, TimeUnit.SECONDS);
    });
    return retrySignal;
}).subscribe(m -> {
    Log.d(TAG, "Message sent!");
}, throwable -> {
    Log.d(TAG, "Failed to send message.");
});
```

Let’s break this down step-by-step:

When the message fails to send, the handler inside `.retryWhen()` would be invoked with an `Observable<Throwable>`. That `Observable` is then zipped with `Observable.range(0, 6)` and the zipper function (i.e. `(t, i) -> i`) provided will simply ignore the `Throwable` and instead emit a `Long`. Using the emitted `Long` values, a transformation via `.flatMap()` is performed to transform `Long` instances into `Observable.timer()` instances. Each `Observable.timer()` contains a delay that is `2^i` seconds, where `i` is the `Long` value emitted upstream, which then signals when resubscription should occur on **sendMessageObservable**.

In short, we have implemented an exponential backoff retry mechanism where reattempting to send a message will be done at most 6 times with delay increments of 1, 2, 4, 8, 16, 32 seconds. If the message continues to fail even after 6 retries, only then would the Observer's `.onError()` be invoked.

The print statements in Logcat should display (assuming sending a message continues to fail).

```
Retrying... delayed by: 1
Retrying... delayed by: 2
Retrying... delayed by: 4
Retrying... delayed by: 8
Retrying... delayed by: 16
Retrying... delayed by: 32
Failed to send message.
```

Undelivered Errors

There are some cases where errors might be considered as *undelivered*. For example, if an `Exception` occurs but no explicit `.onError()` handling was provided on subscription or if `.onError()` was provided but the subscription has been cancelled/disposed. In these scenarios, RxJava tries to do its best to make sure that errors are not lost or swallowed, but instead are reported in a reasonable manner.

Handling Undelivered Errors

`RxJavaPlugins.onError()` is invoked in RxJava internals to handle undelivered error. Essentially, this default `.onError()` behaves as you would expect if the error occurred outside of the context of RxJava–the error's stack trace is printed out and handling is delegated to the `UncaughtExceptionHandler` of the `Thread` where the error occurred.

Alternatively, a default error notification consumer can also be provided to handle undelivered errors via `RxJavaPlugins.setErrorHandler(Consumer<Throwable>)`. Doing so sets a static variable within RxJava so that undelivered errors would be propagated to the provided handler as opposed to using `RxjavaPlugins.onError()`. This approach is preferred as you can pipe the error into your app's logging module.

```
RxJavaPlugins.setErrorHandler(throwable -> {
    Log.e("MyApp", "Received undelivered error: " + throwable.getMessage());
    // ...
});
```

In addition, if the error occurred on the main thread and a handler was not provided in `RxJavaPlugins.setErrorHandler()`, the default behavior of the main thread's `UncaughtExceptionHandler` is to crash the app. By providing a custom handler, this situation can be mitigated.

Bibliography

Anup Cowkur. "Functional Programming for Android Developers–Part 1." *freeCodeCamp*. https://medium.freecodecamp.com/functional-programming-for-android-developers-part-1-a58d40d6e742

Dan Lew. "Multicasting in RxJava." *Dan Lew Codes*. http://blog.danlew.net/2016/06/13/multicasting-in-rxjava/

Dan Lew. "Don't break the chain: Use RxJava's compose() operator." *Dan Lew Codes*. http://blog.danlew.net/2015/03/02/dont-break-the-chain/

Dan Lew. "Error Handling in RxJava." *Dan Lew Codes*. http://blog.danlew.net/2015/12/08/error-handling-in-rxjava/

David Karnok. "Backpressure." *Stack Overflow*. http://stackoverflow.com/documentation/rx-java/2341/backpressure/7701/introduction#t=201705032347091109595

David Karnok. "SubscribeOn and ObserveOn." *Advanced Reactive Java*. http://akarnokd.blogspot.com/2016/03/subscribeon-and-observeon.html

Ray Ozzie. "The Internet Services Disruption." http://scripting.com/disruption/ozzie/TheInternetServicesDisruptio.htm

ReactiveX. *ReactiveX Wiki*. http://reactivex.io/

RxAndroid. *RxJava bindings for Android.* https://github.com/ReactiveX/RxAndroid

RxBinding. *RxJava binding APIs for Android's UI widgets.* https://github.com/JakeWharton/RxBinding

Rx team. "Rx Design Guidelines." https://blogs.msdn.microsoft.com/rxteam/2010/10/28/rx-design-guidelines/

RxJava. *RxJava Wiki*. https://github.com/ReactiveX/RxJava/wiki

Sameer Dhakal. “Howdy RxJava.” *Fuzz*. https://medium.com/fuzz/howdy-rxjava-8f40fef88181

Tomasz Nurkiewickz and Ben Christensen. *Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications*. California: O’Reilly Media, 2016