# [Understanding Reactive types](https://spring.io/blog/2016/04/19/understanding-reactive-types)

Following previous [Reactive Spring](https://spring.io/blog/2016/02/09/reactive-spring) and [Reactor Core 3.0](https://spring.io/blog/2016/03/11/reactor-core-3-0-becomes-a-unified-reactive-foundation-on-java-8) blog posts, I would like to explain why Reactive types are useful and how they compare to other asynchronous types, based on what we have learned while working on the Spring Framework 5 upcoming Reactive support.

## Why using Reactive types?

Reactive types are not intended to allow you to process your requests or data faster, in fact they will introduce a small overhead compared to regular blocking processing. Their strength lies in their capacity to serve more request concurrently, and to handle operations with latency, such as requesting data from a remote server, more efficiently. They allow you to provide a better quality of service and a predictable capacity planning by dealing natively with time and latency without consuming more resources. Unlike traditional processing that blocks the current thread while waiting a result, a Reactive API that waits costs nothing, requests only the amount of data it is able to process and bring new capabilities since it deals with stream of data, not only with individual elements one by one.

## Before Java 8

Before Java 8, asynchronous non-blocking behavior was not obvious to implement for at least two reasons. The first reason is that callback based API required verbose anonymous classes and are not easy to chain. The second reason is that [Future](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html) type is asynchronous **but** blocks the current thread until the computation completes when you try to get the result with the [get()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html#get--) method. That’s why Spring Framework 4.0 introduced [ListenableFuture](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/util/concurrent/ListenableFuture.html), a Future implementation that adds non-blocking callback-based capabilities.

## Lambdas, CompletableFuture and Stream

Then Java 8 introduced lambdas and [CompletableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html). Lambdas allow to write concise callbacks, while [CompletionStage](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html) interface and [CompletableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html) class finally allows to deal with future in a non-blocking way and push-based fashion, while providing capabilities to chain such deferred result processing.

Java 8 also introduced [Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html), which has been designed to deal efficiently with stream of data (including primitive types) that can be accessed with no or very little latency. It is pull-based, can only be used once, lacks time-related operations and can perform parallel computations but without being able to specify the thread pool to use. As explained by [Brian Goetz](https://stackoverflow.com/questions/17960656/is-it-possible-to-use-java-8-streams-api-for-asynchronous-processing/18826615#comment36545451_18826615),  
it has not been designed to deal with operation with latency, such as I/O operations. And that is where Reactive APIs like Reactor or RxJava come in.

## Reactive APIs

Reactive APIs such as [Reactor](https://projectreactor.io/) also provide operators like Java 8 Stream, but they work more generally with any stream sequence (not just Collections) and allow to define a pipeline of transforming operations that will apply to the data passing through it thanks to a handy fluent API and using lambdas. They are designed to handle both synchronous or asynchronous operations, and allow you to buffer, merge, concatenate, or apply a wide range of transformations to your data.

Initially Reactive APIs were only designed to deal with streams of data, i.e. N elements, for example, using Reactor’s [Flux](http://projectreactor.io/core/docs/api/reactor/core/publisher/Flux.html):

reactiveService.getResults()

.mergeWith(Flux.interval(100))

.doOnNext(serviceA::someObserver)

.map(d -> d \* 2)

.take(3)

.onErrorResumeWith(errorHandler::fallback)

.doAfterTerminate(serviceM::incrementTerminate)

.consume(System.out::println);

But during our work on Spring Framework 5, it became apparent that there was a clear need to distinguish between streams of 1 or N elements, and that is why Reactor provides the [Mono](http://projectreactor.io/core/docs/api/reactor/core/publisher/Mono.html) type. Mono is the Reactive equivalent of CompletableFuture type, and allow to provide a consistent API for handling single and multiple elements in a Reactive way.

Mono.any(reactiveServiceA.findRecent(time), reactiveServiceB.findRecent(time)

.timeout(Duration.ofSeconds(3), errorHandler::fallback)

.doOnSuccess(r -> reactiveServiceC.incrementSuccess())

.consume(System.out::println);

If you have a deeper look to [Flux](http://projectreactor.io/core/docs/api/reactor/core/publisher/Flux.html) and [Mono](http://projectreactor.io/core/docs/api/reactor/core/publisher/Mono.html), you will notice these types implement the Publisher interface from the Reactive Streams specification.

## Reactive Streams

Reactor is built on the [Reactive Streams](http://www.reactive-streams.org) specification. Reactive Streams is composed of [4 simple Java interfaces](http://www.reactive-streams.org/reactive-streams-1.0.0-javadoc/) (Publisher, Subscriber, Subscription and Processor), a [textual specification](https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.0/README.md#specification) and a [TCK](http://www.reactive-streams.org/reactive-streams-tck-1.0.0-javadoc/). It is the cornerstone of every modern Reactive library and a must have for interoperability purpose.

The core concern of Reactive Streams is handling backpressure. In a nutshell, backpressure is a mechanism that permits a receiver to ask how much data it wants to receive from the emitter. It allows:

* The receiver to start receiving data only when it is ready to process it
* To control the inflight amount of data
* Efficient handling of slow emitter/fast receiver or fast emitter/slow receiver use cases
* To switch from a dynamic push-pull strategy to a push-based only strategy if you request Long.MAX\_VALUE elements

At first glance, the [Publisher](http://www.reactive-streams.org/reactive-streams-1.0.0-javadoc/org/reactivestreams/Publisher.html) interface seems deceivingly simple to implement; but doing so in complete conformance with the specification turns out to be pretty hard, and users can’t do anything with raw Publisher except subscribing to it! That’s why it’s typically a better idea to rely on a Reactive Streams implementation, such as Reactor, to help you out with this.

Note that Java 9 [will include](http://cs.oswego.edu/pipermail/concurrency-interest/2015-January/013641.html) the Reactive Streams interfaces in the java.util.concurrent.Flow container class, further showing the relevance of Reactive Streams within the JDK.

It is also important to notice that convergence toward Reactive Streams and [Reactor conversion capabilities](https://github.com/reactor/reactor-core/tree/master/src/main/java/reactor/core/converter) allow easy and efficient conversion from one Reactive type to another at runtime.

## Conclusion

I hope this blog post will help you to have a better understanding of Reactive types.

We are working on Reactive support with types like Reactor Mono and Flux in various Spring projects like Spring Framework, Spring Boot, Spring Data, Spring Security and Spring Cloud.

But your upcoming Reactive application will also use directly these types too, for example at @Repository, @Service or @Controller methods level, because building a Reactive application means using Reactive semantics where you have to deal with latency or streams (we will also provide some guidance to integrate blocking API).

We will post additional Reactive blog posts in the upcoming months. Feel free to familiarize yourself with [this test-driven Lite Rx API Hands-On](https://github.com/reactor/lite-rx-api-hands-on/) that will teach you how to use Flux and Mono, and as usual your feedbacks are welcome!

If you happen to be in Barcelona mid May (never a bad time to be in Barcelona anyway!), don’t miss the chance to join the [Spring I/O conference](http://www.springio.net/). Also, the registration for [SpringOne Platform](http://springoneplatform.io/) (early August, Las Vegas) has opened recently, in case you want to benefit from early bird ticket pricing.

[**Notes on Reactive Programming Part I: The Reactive Landscape**](https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape)

Reactive Programming is interesting (again) and there is a lot of noise about it at the moment, not all of which is very easy to understand for an outsider and simple enterprise Java developer, such as the author. This article (the first in a series) might help to clarify your understanding of what the fuss is about. The approach is as concrete as possible, and there is no mention of "denotational semantics". If you are looking for a more academic approach and loads of code samples in Haskell, the internet is full of them, but you probably don’t want to be here.

Reactive Programming is often conflated with concurrent programming and high performance to such an extent that it’s hard to separate those concepts, when actually they are in principle completely different. This inevitably leads to confusion. Reactive Programming is also often referred to as or conflated with Functional Reactive Programming, or FRP (and we use the two interchangeably here). Some people think Reactive is nothing new, and it’s what they do all day anyway (mostly they use JavaScript). Others seem to think that it’s a gift from Microsoft (who made a big splash about it when they released some C# extensions a while ago). In the Enterprise Java space there has been something of a buzz recently (e.g. see the [Reactive Streams initiative](http://www.reactive-streams.org/)), and as with anything shiny and new, there are a lot of easy mistakes to make out there, about when and where it can and should be used.

**What Is It?**

Reactive Programming is a style of micro-architecture involving intelligent routing and consumption of events, all combining to change behaviour. That’s a bit abstract, and so are many of the other definitions you will come across online. We attempt build up some more concrete notions of what it means to be reactive, or why it might be important in what follows.

The origins of Reactive Programming can probably be traced to the 1970s or even earlier, so there’s nothing new about the idea, but they are really resonating with something in the modern enterprise. This resonance has arrived (not accidentally) at the same time as the rise of microservices, and the ubiquity of multi-core processors. Some of the reasons for that will hopefully become clear.

Here are some useful potted definitions from other sources:

The basic idea behind reactive programming is that there are certain

datatypes that represent a value "over time". Computations that

involve these changing-over-time values will themselves have values

that change over time.

and…​

An easy way of reaching a first intuition about what it's like is to

imagine your program is a spreadsheet and all of your variables are

cells. If any of the cells in a spreadsheet change, any cells that

refer to that cell change as well. It's just the same with FRP. Now

imagine that some of the cells change on their own (or rather, are

taken from the outside world): in a GUI situation, the position of

the mouse would be a good example.

(from [Terminology Question on Stackoverflow](https://stackoverflow.com/questions/1028250/what-is-functional-reactive-programming))

FRP has a strong affinity with high-performance, concurrency, asynchronous operations and non-blocking IO. However, it might be helpful to start with a suspicion that FRP has nothing to do with any of them. It is certainly the case that such concerns can be naturally handled, often transparently to the caller, when using a Reactive model. But the actual benefit, in terms of handling those concerns effectively or efficiently is entirely up to the implementation in question (and therefore should be subject to a high degree of scrutiny). It is also possible to implement a perfectly sane and useful FRP framework in a synchronous, single-threaded way, but that isn’t really likely to be helpful in trying to use any of the new tools and libraries.

**Reactive Use Cases**

The hardest question to get an answer to as a newbie seems to be "what is it good for?" Here are some examples from an enterprise setting that illustrate general patterns of use:

**External Service Calls** Many backend services these days are REST-ful (i.e. they operate over HTTP) so the underlying protocol is fundamentally blocking and synchronous. Not obvious territory for FRP maybe, but actually it’s quite fertile ground because the implementation of such services often involves calling other services, and then yet more services depending on the results from the first calls. With so much IO going on if you were to wait for one call to complete before sending the next request, your poor client would give up in frustration before you managed to assemble a reply. So external service calls, especially complex orchestrations of dependencies between calls, are a good thing to optimize. FRP offers the promise of "composability" of the logic driving those operations, so that it is easier to write for the developer of the calling service.

**Highly Concurrent Message Consumers** Message processing, in particular when it is highly concurrent, is a common enterprise use case. Reactive frameworks like to measure micro benchmarks, and brag about how many messages per second you can process in the JVM. The results are truly staggering (tens of millions of messages per second are easy to achieve), but possibly somewhat artificial - you wouldn’t be so impressed if they said they were benchmarking a simple "for" loop. However, we should not be too quick to write off such work, and it’s easy to see that when performance matters, all contributions should be gratefully accepted. Reactive patterns fit naturally with message processing (since an event translates nicely into a message), so if there is a way to process more messages faster we should pay attention.

**Spreadsheets** Perhaps not really an enterprise use case, but one that everyone in the enterprise can easily relate to, and it nicely captures the philosophy of, and difficulty of implementing FRP. If cell B depends on cell A, and cell C depends on both cells A and B, then how do you propagate changes in A, ensuring that C is updated before any change events are sent to B? If you have a truly reactive framework to build on, then the answer is "you don’t care, you just declare the dependencies," and that is really the power of a spreadsheet in a nutshell. It also highlights the difference between FRP and simple event-driven programming — it puts the "intelligent" in "intelligent routing".

**Abstraction Over (A)synchronous Processing** This is more of an abstract use case, so straying into the territory we should perhaps be avoiding. There is also some (a lot) of overlap between this and the more concrete use cases already mentioned, but hopefully it is still worth some discussion. The basic claim is a familiar (and justifiable) one, that as long as developers are willing to accept an extra layer of abstraction, they can forget about whether the code they are calling is synchronous or asynchronous. Since it costs precious brain cells to deal with asynchronous programming, there could be some useful ideas there. Reactive Programming is not the only approach to this issue, but some of the implementaters of FRP have thought hard enough about this problem that their tools are useful.

This Netflix blog has some really useful concrete examples of real-life use cases: [Netflix Tech Blog: Functional Reactive in the Netflix API with RxJava](http://techblog.netflix.com/2013/02/rxjava-netflix-api.html)

**Comparisons**

If you haven’t been living in a cave since 1970 you will have come across some other concepts that are relevant to Reactive Programming and the kinds of problems people try and solve with it. Here are a few of them with my personal take on their relevance:

**Ruby Event-Machine** The [Event Machine](https://github.com/eventmachine/eventmachine) is an abstraction over concurrent programming (usually involving non-blocking IO). Rubyists struggled for a long time to turn a language that was designed for single-threaded scripting into something that you could use to write a server application that a) worked, b) performed well, and c) stayed alive under load. Ruby has had threads for quite some time, but they aren’t used much and have a bad reputation because they don’t always perform very well. The alternative, which is ubiquitous now that it has been promoted (in Ruby 1.9) to the core of the language, is [Fibers](https://www.ruby-doc.org/core-1.9.3/Fiber.html)(sic). The Fiber programming model is sort of a flavour of coroutines (see below), where a single native thread is used to process large numbers of concurrent requests (usually involving IO). The programming model itself is a bit abstract and hard to reason about, so most people use a wrapper, and the Event Machine is the most common. Event Machine doesn’t necessarily use Fibers (it abstracts those concerns), but it is easy to find examples of code using Event Machine with Fibers in Ruby web apps (e.g. [see this article by Ilya Grigorik](https://www.igvita.com/2009/05/13/fibers-cooperative-scheduling-in-ruby), or the [fibered example from em-http-request](https://github.com/igrigorik/em-http-request/blob/master/examples/fibered-http.rb)). People do this a lot to get the benefit of scalability that comes from using Event Machine in an I/O intensive application, without the ugly programming model that you get with lots of nested callbacks.

**Actor Model** Similar to Object Oriented Programming, the Actor Model is a deep thread of Computer Science going back to the 1970s. Actors provide an abstraction over computation (as opposed to data and behaviour) that allows for concurrency as a natural consequence, so in practical terms they can form the basis of a concurrent system. Actors send each other messages, so they are reactive in some sense, and there is a lot of overlap between systems that style themselves as Actors or Reactive. Often the distinction is at the level of their implementation (e.g. Actors in [Akka](http://doc.akka.io/docs/akka/current/java.html) can be distributed across processes, and that is a distinguishing feature of that framework).

**Deferred results (Futures)** Java 1.5 introduced a rich new set of libraries including Doug Lea’s "java.util.concurrent", and part of that is the concept of a deferred result, encapsulated in a Future. It’s a good example of a simple abstraction over an asynchronous pattern, without forcing the implementation to be asynchronous, or use any particular model of asynchronous processing. As the [Netflix Tech Blog: Functional Reactive in the Netflix API with RxJava](http://techblog.netflix.com/2013/02/rxjava-netflix-api.html) shows nicely, Futures are great when all you need is concurrent processing of a set of similar tasks, but as soon as any of them want to depend on each other or execute conditionally you get into a form of "nested callback hell". Reactive Programming provides an antidote to that.

**Map-reduce and fork-join** Abstractions over parallel processing are useful and there are many examples to choose from. Map-reduce and fork-join that have evolved recently in the Java world, driven by massively parallel distributed processing ([MapReduce](http://research.google.com/archive/mapreduce-osdi04.pdf) and [Hadoop](https://wiki.apache.org/hadoop/MapReduce)) and by the JDK itself in version 1.7 ([Fork-Join](http://gee.cs.oswego.edu/dl/papers/fj.pdf)). These are useful abstractions but (like deferred results) they are shallow compared to FRP, which can be used as an abstraction over simple parallel processing, but which reaches beyond that into composability and declarative communication.

**Coroutines** A ["coroutine"](https://en.wikipedia.org/wiki/Coroutines) is a generalization of a "subroutine" — it has an entry point, and exit point(s) like a subroutine, but when it exits it passes control to another coroutine (not necessarily to its caller), and whatever state it accumulated is kept and remembered for the next time it is called. Coroutines can be used as a building block for higher level features like Actors and Streams. One of the goals of Reactive Programming is to provide the same kind of abstraction over communicating parallel processing agents, so coroutines (if they are available) are a useful building block. There are various flavours of coroutines, some of which are more restrictive than the general case, but more flexible than vanilla subroutines. Fibers (see the discussion on Event Machine) are one flavour, and Generators (familiar in Scala and Python) are another.

**Reactive Programming in Java**

Java is not a "reactive language" in the sense that it doesn’t support coroutines natively. There are other languages on the JVM (Scala and Clojure) that support reactive models more natively, but Java itself does not until version 9. Java, however, is a powerhouse of enterprise development, and there has been a lot of activity recently in providing Reactive layers on top of the JDK. We only take a very brief look at a few of them here.

[**Reactive Streams**](http://www.reactive-streams.org/) is a very low level contract, expressed as a handful of Java interfaces (plus a TCK), but also applicable to other languages. The interfaces express the basic building blocks of Publisher and Subscriber with explicit back pressure, forming a common language for interoperable libraries. Reactive Streams have been incorporated into the JDK as java.util.concurrent.Flow in version 9. The project is a collaboration between engineers from Kaazing, Netflix, Pivotal, Red Hat, Twitter, Typesafe and many others.

[**RxJava**](https://github.com/ReactiveX/RxJava/wiki): Netflix were using reactive patterns internally for some time and then they released the tools they were using under an open source license as [Netflix/RxJava](https://github.com/ReactiveX/RxJava/wiki) (subsequently re-branded as "ReactiveX/RxJava"). Netflix does a lot of programming in Groovy on top of RxJava, but it is open to Java usage and quite well suited to Java 8 through the use of Lambdas. There is a [bridge to Reactive Streams](https://github.com/ReactiveX/RxJavaReactiveStreams). RxJava is a "2nd Generation" library according to David Karnok’s [Generations of Reactive](https://akarnokd.blogspot.co.uk/2016/03/operator-fusion-part-1.html) classification.

[**Reactor**](https://projectreactor.io/) is a Java framework from the [Pivotal](http://www.gopivotal.com/oss) open source team (the one that created Spring). It builds directly on Reactive Streams, so there is no need for a bridge. The Reactor IO project provides wrappers around low-level network runtimes like Netty and Aeron. Reactor is a "4th Generation" library according to David Karnok’s [Generations of Reactive](https://akarnokd.blogspot.co.uk/2016/03/operator-fusion-part-1.html) classification.

[**Spring Framework 5.0**](https://projects.spring.io/spring-framework/) (first milestone June 2016) has reactive features built into it, including tools for building HTTP servers and clients. Existing users of Spring in the web tier will find a very familiar programming model using annotations to decorate controller methods to handle HTTP requests, for the most part handing off the dispatching of reactive requests and back pressure concerns to the framework. Spring builds on Reactor, but also exposes APIs that allow its features to be expressed using a choice of libraries (e.g. Reactor or RxJava). Users can choose from Tomcat, Jetty, Netty (via Reactor IO) and Undertow for the server side network stack.

[**Ratpack**](https://ratpack.io) is a set of libraries for building high performance services over HTTP. It builds on Netty and implements Reactive Streams for interoperability (so you can use other Reactive Streams implementations higher up the stack, for instance). Spring is supported as a native component, and can be used to provide dependency injection using some simple utility classes. There is also some autoconfiguration so that Spring Boot users can embed Ratpack inside a Spring application, bringing up an HTTP endpoint and listening there instead of using one of the embedded servers supplied directly by Spring Boot.

[**Akka**](http://akka.io/) is a toolkit for building applications using the Actor pattern in Scala or Java, with interprocess communication using Akka Streams, and Reactive Streams contracts are built in. Akka is a "3rd Generation" library according to David Karnok’s [Generations of Reactive](https://akarnokd.blogspot.co.uk/2016/03/operator-fusion-part-1.html) classification.

**Why Now?**

What is driving the rise of Reactive in Enterprise Java? Well, it’s not (all) just a technology fad — people jumping on the bandwagon with the shiny new toys. The driver is efficient resource utilization, or in other words, spending less money on servers and data centres. The promise of Reactive is that you can do more with less, specifically you can process higher loads with fewer threads. This is where the intersection of Reactive and non-blocking, asynchronous I/O comes to the foreground. For the right problem, the effects are dramatic. For the wrong problem, the effects might go into reverse (you actually make things worse). Also remember, even if you pick the right problem, there is no such thing as a free lunch, and Reactive doesn’t solve the problems for you, it just gives you a toolbox that you can use to implement solutions.

**Conclusion**

In this article we have taken a very broad and high level look at the Reactive movement, setting it in context in the modern enterprise. There are a number of Reactive libraries or frameworks for the JVM, all under active development. To a large extent they provide similar features, but increasingly, thanks to Reactive Streams, they are interoperable. In the [next article](https://spring.io/blog/2016/06/13/notes-on-reactive-programming-part-ii-writing-some-code) in the series we will get down to brass tacks and have a look at some actual code samples, to get a better picture of the specifics of what it means to be Reactive and why it matters. We will also devote some time to understanding why the "F" in FRP is important, and how the concepts of back pressure and non-blocking code have a profound impact on programming style. And most importantly, we will help you to make the important decision about when and how to go Reactive, and when to stay put on the older styles and stacks.

## [The introduction to Reactive Programming you've been missing](https://gist.github.com/staltz/868e7e9bc2a7b8c1f754)

(by [@andrestaltz](https://twitter.com/andrestaltz))

### This tutorial as a series of videos

**If you prefer to watch video tutorials with live-coding, then check out this series I recorded with the same contents as in this article:** [**Egghead.io - Introduction to Reactive Programming**](https://egghead.io/series/introduction-to-reactive-programming)**.**

So you're curious in learning this new thing called Reactive Programming, particularly its variant comprising of Rx, Bacon.js, RAC, and others.

Learning it is hard, even harder by the lack of good material. When I started, I tried looking for tutorials. I found only a handful of practical guides, but they just scratched the surface and never tackled the challenge of building the whole architecture around it. Library documentations often don't help when you're trying to understand some function. I mean, honestly, look at this:

**Rx.Observable.prototype.flatMapLatest(selector, [thisArg])**

Projects each element of an observable sequence into a new sequence of observable sequences by incorporating the element's index and then transforms an observable sequence of observable sequences into an observable sequence producing values only from the most recent observable sequence.

Holy cow.

I've read two books, one just painted the big picture, while the other dived into how to use the Reactive library. I ended up learning Reactive Programming the hard way: figuring it out while building with it. At my work in [Futurice](https://www.futurice.com) I got to use it in a real project, and had the [support of some colleagues](http://blog.futurice.com/top-7-tips-for-rxjava-on-android) when I ran into troubles.

The hardest part of the learning journey is **thinking in Reactive**. It's a lot about letting go of old imperative and stateful habits of typical programming, and forcing your brain to work in a different paradigm. I haven't found any guide on the internet in this aspect, and I think the world deserves a practical tutorial on how to think in Reactive, so that you can get started. Library documentation can light your way after that. I hope this helps you.

## "What is Reactive Programming?"

There are plenty of bad explanations and definitions out there on the internet. [Wikipedia](https://en.wikipedia.org/wiki/Reactive_programming) is too generic and theoretical as usual. [Stackoverflow](http://stackoverflow.com/questions/1028250/what-is-functional-reactive-programming)'s canonical answer is obviously not suitable for newcomers. [Reactive Manifesto](http://www.reactivemanifesto.org/) sounds like the kind of thing you show to your project manager or the businessmen at your company. Microsoft's [Rx terminology](https://rx.codeplex.com/) "Rx = Observables + LINQ + Schedulers" is so heavy and Microsoftish that most of us are left confused. Terms like "reactive" and "propagation of change" don't convey anything specifically different to what your typical MV\* and favorite language already does. Of course my framework views react to the models. Of course change is propagated. If it wouldn't, nothing would be rendered.

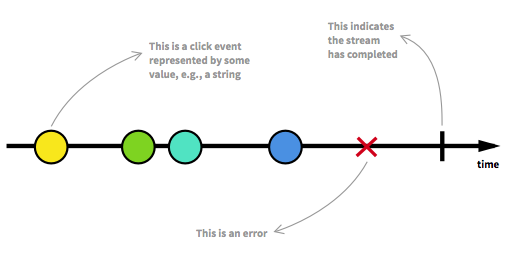
So let's cut the bullshit.

#### Reactive programming is programming with asynchronous data streams.

In a way, this isn't anything new. Event buses or your typical click events are really an asynchronous event stream, on which you can observe and do some side effects. Reactive is that idea on steroids. You are able to create data streams of anything, not just from click and hover events. Streams are cheap and ubiquitous, anything can be a stream: variables, user inputs, properties, caches, data structures, etc. For example, imagine your Twitter feed would be a data stream in the same fashion that click events are. You can listen to that stream and react accordingly.

**On top of that, you are given an amazing toolbox of functions to combine, create and filter any of those streams.** That's where the "functional" magic kicks in. A stream can be used as an input to another one. Even multiple streams can be used as inputs to another stream. You can merge two streams. You can filter a stream to get another one that has only those events you are interested in. You can map data values from one stream to another new one.

If streams are so central to Reactive, let's take a careful look at them, starting with our familiar "clicks on a button" event stream.

[](https://camo.githubusercontent.com/36c0a9ffd8ed22236bd6237d44a1d3eecbaec336/687474703a2f2f692e696d6775722e636f6d2f634c344d4f73532e706e67)

A stream is a sequence of **ongoing events ordered in time**. It can emit three different things: a value (of some type), an error, or a "completed" signal. Consider that the "completed" takes place, for instance, when the current window or view containing that button is closed.

We capture these emitted events only **asynchronously**, by defining a function that will execute when a value is emitted, another function when an error is emitted, and another function when 'completed' is emitted. Sometimes these last two can be omitted and you can just focus on defining the function for values. The "listening" to the stream is called **subscribing**. The functions we are defining are observers. The stream is the subject (or "observable") being observed. This is precisely the [Observer Design Pattern](https://en.wikipedia.org/wiki/Observer_pattern).

An alternative way of drawing that diagram is with ASCII, which we will use in some parts of this tutorial:

--a---b-c---d---X---|->

a, b, c, d are emitted values

X is an error

| is the 'completed' signal

---> is the timeline

Since this feels so familiar already, and I don't want you to get bored, let's do something new: we are going to create new click event streams transformed out of the original click event stream.

First, let's make a counter stream that indicates how many times a button was clicked. In common Reactive libraries, each stream has many functions attached to it, such as map, filter, scan, etc. When you call one of these functions, such as clickStream.map(f), it returns a **new stream** based on the click stream. It does not modify the original click stream in any way. This is a property called **immutability**, and it goes together with Reactive streams just like pancakes are good with syrup. That allows us to chain functions like clickStream.map(f).scan(g):

clickStream: ---c----c--c----c------c-->

vvvvv map(c becomes 1) vvvv

---1----1--1----1------1-->

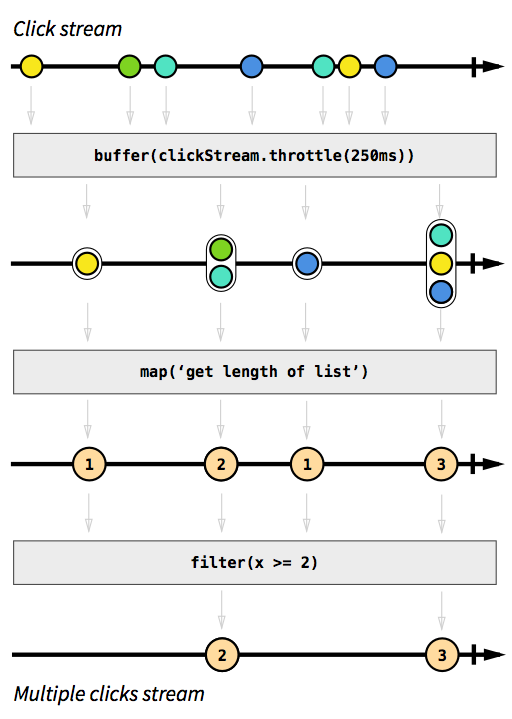
vvvvvvvvv scan(+) vvvvvvvvv

counterStream: ---1----2--3----4------5-->

The map(f) function replaces (into the new stream) each emitted value according to a function f you provide. In our case, we mapped to the number 1 on each click. The scan(g) function aggregates all previous values on the stream, producing value x = g(accumulated, current), where g was simply the add function in this example. Then, counterStream emits the total number of clicks whenever a click happens.

To show the real power of Reactive, let's just say that you want to have a stream of "double click" events. To make it even more interesting, let's say we want the new stream to consider triple clicks as double clicks, or in general, multiple clicks (two or more). Take a deep breath and imagine how you would do that in a traditional imperative and stateful fashion. I bet it sounds fairly nasty and involves some variables to keep state and some fiddling with time intervals.

Well, in Reactive it's pretty simple. In fact, the logic is just [4 lines of code](http://jsfiddle.net/staltz/4gGgs/27/). But let's ignore code for now. Thinking in diagrams is the best way to understand and build streams, whether you're a beginner or an expert.

[](https://camo.githubusercontent.com/995c301de2f566db10748042a5a67cc5d9ac45d9/687474703a2f2f692e696d6775722e636f6d2f484d47574e4f352e706e67)

Grey boxes are functions transforming one stream into another. First we accumulate clicks in lists, whenever 250 milliseconds of "event silence" has happened (that's what buffer(stream.throttle(250ms)) does, in a nutshell. Don't worry about understanding the details at this point, we are just demoing Reactive for now). The result is a stream of lists, from which we apply map() to map each list to an integer matching the length of that list. Finally, we ignore 1 integers using the filter(x >= 2) function. That's it: 3 operations to produce our intended stream. We can then subscribe ("listen") to it to react accordingly how we wish.

I hope you enjoy the beauty of this approach. This example is just the tip of the iceberg: you can apply the same operations on different kinds of streams, for instance, on a stream of API responses; on the other hand, there are many other functions available.

## "Why should I consider adopting RP?"

Reactive Programming raises the level of abstraction of your code so you can focus on the interdependence of events that define the business logic, rather than having to constantly fiddle with a large amount of implementation details. Code in RP will likely be more concise.

The benefit is more evident in modern webapps and mobile apps that are highly interactive with a multitude of UI events related to data events. 10 years ago, interaction with web pages was basically about submitting a long form to the backend and performing simple rendering to the frontend. Apps have evolved to be more real-time: modifying a single form field can automatically trigger a save to the backend, "likes" to some content can be reflected in real time to other connected users, and so forth.

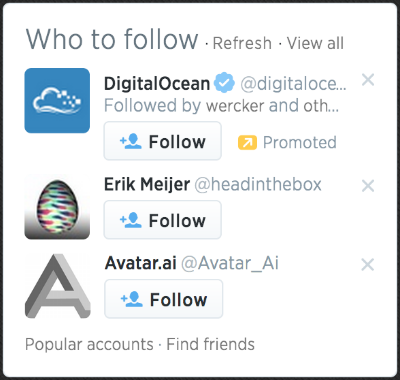
Apps nowadays have an abundancy of real-time events of every kind that enable a highly interactive experience to the user. We need tools for properly dealing with that, and Reactive Programming is an answer.

## Thinking in RP, with examples

Let's dive into the real stuff. A real-world example with a step-by-step guide on how to think in RP. No synthetic examples, no half-explained concepts. By the end of this tutorial we will have produced real functioning code, while knowing why we did each thing.

I picked **JavaScript** and [**RxJS**](https://github.com/Reactive-Extensions/RxJS) as the tools for this, for a reason: JavaScript is the most familiar language out there at the moment, and the [Rx\* library family](http://www.reactivex.io) is widely available for many languages and platforms ([.NET](https://rx.codeplex.com/), [Java](https://github.com/Netflix/RxJava), [Scala](https://github.com/Netflix/RxJava/tree/master/language-adaptors/rxjava-scala), [Clojure](https://github.com/Netflix/RxJava/tree/master/language-adaptors/rxjava-clojure), [JavaScript](https://github.com/Reactive-Extensions/RxJS), [Ruby](https://github.com/Reactive-Extensions/Rx.rb), [Python](https://github.com/Reactive-Extensions/RxPy), [C++](https://github.com/Reactive-Extensions/RxCpp), [Objective-C/Cocoa](https://github.com/ReactiveCocoa/ReactiveCocoa), [Groovy](https://github.com/Netflix/RxJava/tree/master/language-adaptors/rxjava-groovy), etc). So whatever your tools are, you can concretely benefit by following this tutorial.

## Implementing a "Who to follow" suggestions box

[](https://camo.githubusercontent.com/81e5d63c69768e1b04447d2e246f47540dd83fbd/687474703a2f2f692e696d6775722e636f6d2f65416c4e62306a2e706e67)In Twitter there is this UI element that suggests other accounts you could follow:

We are going to focus on imitating its core features, which are:

* On startup, load accounts data from the API and display 3 suggestions
* On clicking "Refresh", load 3 other account suggestions into the 3 rows
* On click 'x' button on an account row, clear only that current account and display another
* Each row displays the account's avatar and links to their page

We can leave out the other features and buttons because they are minor. And, instead of Twitter, which recently closed its API to the unauthorized public, let's build that UI for following people on Github. There's a [Github API for getting users](https://developer.github.com/v3/users/#get-all-users).

The complete code for this is ready at <http://jsfiddle.net/staltz/8jFJH/48/> in case you want to take a peak already.

## Request and response

**How do you approach this problem with Rx?** Well, to start with, (almost) everything can be a stream. That's the Rx mantra. Let's start with the easiest feature: "on startup, load 3 accounts data from the API". There is nothing special here, this is simply about (1) doing a request, (2) getting a response, (3) rendering the response. So let's go ahead and represent our requests as a stream. At first this will feel like overkill, but we need to start from the basics, right?

On startup we need to do only one request, so if we model it as a data stream, it will be a stream with only one emitted value. Later, we know we will have many requests happening, but for now, it is just one.

--a------|->

Where a is the string 'https://api.github.com/users'

This is a stream of URLs that we want to request. Whenever a request event happens, it tells us two things: when and what. "When" the request should be executed is when the event is emitted. And "what" should be requested is the value emitted: a string containing the URL.

To create such stream with a single value is very simple in Rx\*. The official terminology for a stream is "Observable", for the fact that it can be observed, but I find it to be a silly name, so I call it stream.

var requestStream = Rx.Observable.just('https://api.github.com/users');

But now, that is just a stream of strings, doing no other operation, so we need to somehow make something happen when that value is emitted. That's done by [subscribing](https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md#rxobservableprototypesubscribeobserver--onnext-onerror-oncompleted) to the stream.

requestStream.subscribe(function(requestUrl) {

// execute the request

jQuery.getJSON(requestUrl, function(responseData) {

// ...

});

}

Notice we are using a jQuery Ajax callback (which we assume you [should know already](http://devdocs.io/jquery/jquery.getjson)) to handle the asynchronicity of the request operation. But wait a moment, Rx is for dealing with **asynchronous** data streams. Couldn't the response for that request be a stream containing the data arriving at some time in the future? Well, at a conceptual level, it sure looks like it, so let's try that.

requestStream.subscribe(function(requestUrl) {

// execute the request

var responseStream = Rx.Observable.create(function (observer) {

jQuery.getJSON(requestUrl)

.done(function(response) { observer.onNext(response); })

.fail(function(jqXHR, status, error) { observer.onError(error); })

.always(function() { observer.onCompleted(); });

});

responseStream.subscribe(function(response) {

// do something with the response

});

}

What [Rx.Observable.create()](https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md#rxobservablecreatesubscribe) does is create your own custom stream by explicitly informing each observer (or in other words, a "subscriber") about data events (onNext()) or errors (onError()). What we did was just wrap that jQuery Ajax Promise. **Excuse me, does this mean that a Promise is an Observable?**

[Amazed](https://camo.githubusercontent.com/4df519edd2d527bf5e90b7d00e22cdc3c3be00d4/687474703a2f2f7777772e6d79666163657768656e2e6e65742f75706c6f6164732f333332342d616d617a65642d666163652e676966)Yes.

Observable is Promise++. In Rx you can easily convert a Promise to an Observable by doing var stream = Rx.Observable.fromPromise(promise), so let's use that. The only difference is that Observables are not [Promises/A+](http://promises-aplus.github.io/promises-spec/) compliant, but conceptually there is no clash. A Promise is simply an Observable with one single emitted value. Rx streams go beyond promises by allowing many returned values.

This is pretty nice, and shows how Observables are at least as powerful as Promises. So if you believe the Promises hype, keep an eye on what Rx Observables are capable of.

Now back to our example, if you were quick to notice, we have one subscribe() call inside another, which is somewhat akin to callback hell. Also, the creation of responseStream is dependent on requestStream. As you heard before, in Rx there are simple mechanisms for transforming and creating new streams out of others, so we should be doing that.

The one basic function that you should know by now is [map(f)](https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md#rxobservableprototypemapselector-thisarg), which takes each value of stream A, applies f() on it, and produces a value on stream B. If we do that to our request and response streams, we can map request URLs to response Promises (disguised as streams).

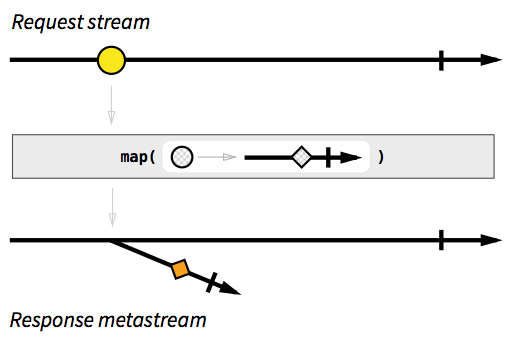
var responseMetastream = requestStream

.map(function(requestUrl) {

return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));

});

Then we will have created a beast called "metastream": a stream of streams. Don't panic yet. A metastream is a stream where each emitted value is yet another stream. You can think of it as [pointers](https://en.wikipedia.org/wiki/Pointer_%28computer_programming%29): each emitted value is a pointer to another stream. In our example, each request URL is mapped to a pointer to the promise stream containing the corresponding response.

[](https://camo.githubusercontent.com/2a8a9cc75acd13443f588fd7f386bd7a6dcb271a/687474703a2f2f692e696d6775722e636f6d2f48486e6d6c61632e706e67)

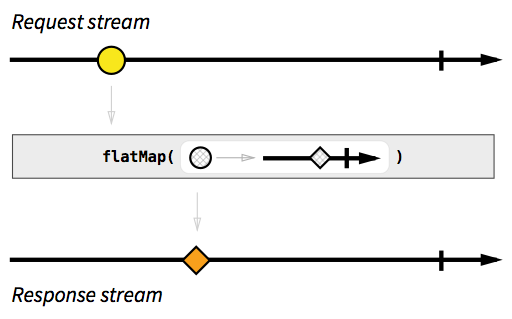
A metastream for responses looks confusing, and doesn't seem to help us at all. We just want a simple stream of responses, where each emitted value is a JSON object, not a 'Promise' of a JSON object. Say hi to [Mr. Flatmap](https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md#rxobservableprototypeflatmapselector-resultselector): a version of map() that "flattens" a metastream, by emitting on the "trunk" stream everything that will be emitted on "branch" streams. Flatmap is not a "fix" and metastreams are not a bug, these are really the tools for dealing with asynchronous responses in Rx.

var responseStream = requestStream

.flatMap(function(requestUrl) {

return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));

});

[](https://camo.githubusercontent.com/0b0ac4a249e1c15d7520c220957acfece1af3e95/687474703a2f2f692e696d6775722e636f6d2f4869337a4e7a4a2e706e67)

Nice. And because the response stream is defined according to request stream, **if** we have later on more events happening on request stream, we will have the corresponding response events happening on response stream, as expected:

requestStream: --a-----b--c------------|->

responseStream: -----A--------B-----C---|->

(lowercase is a request, uppercase is its response)

Now that we finally have a response stream, we can render the data we receive:

responseStream.subscribe(function(response) {

// render `response` to the DOM however you wish

});

Joining all the code until now, we have:

var requestStream = Rx.Observable.just('https://api.github.com/users');

var responseStream = requestStream

.flatMap(function(requestUrl) {

return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));

});

responseStream.subscribe(function(response) {

// render `response` to the DOM however you wish

});

## The refresh button

I did not yet mention that the JSON in the response is a list with 100 users. The API only allows us to specify the page offset, and not the page size, so we're using just 3 data objects and wasting 97 others. We can ignore that problem for now, since later on we will see how to cache the responses.

Everytime the refresh button is clicked, the request stream should emit a new URL, so that we can get a new response. We need two things: a stream of click events on the refresh button (mantra: anything can be a stream), and we need to change the request stream to depend on the refresh click stream. Gladly, RxJS comes with tools to make Observables from event listeners.

var refreshButton = document.querySelector('.refresh');

var refreshClickStream = Rx.Observable.fromEvent(refreshButton, 'click');

Since the refresh click event doesn't itself carry any API URL, we need to map each click to an actual URL. Now we change the request stream to be the refresh click stream mapped to the API endpoint with a random offset parameter each time.

var requestStream = refreshClickStream

.map(function() {

var randomOffset = Math.floor(Math.random()\*500);

return 'https://api.github.com/users?since=' + randomOffset;

});

Because I'm dumb and I don't have automated tests, I just broke one of our previously built features. A request doesn't happen anymore on startup, it happens only when the refresh is clicked. Urgh. I need both behaviors: a request when either a refresh is clicked or the webpage was just opened.

We know how to make a separate stream for each one of those cases:

var requestOnRefreshStream = refreshClickStream

.map(function() {

var randomOffset = Math.floor(Math.random()\*500);

return 'https://api.github.com/users?since=' + randomOffset;

});

var startupRequestStream = Rx.Observable.just('https://api.github.com/users');

But how can we "merge" these two into one? Well, there's [merge()](https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md#rxobservableprototypemergemaxconcurrent--other). Explained in the diagram dialect, this is what it does:

stream A: ---a--------e-----o----->

stream B: -----B---C-----D-------->

vvvvvvvvv merge vvvvvvvvv

---a-B---C--e--D--o----->

It should be easy now:

var requestOnRefreshStream = refreshClickStream

.map(function() {

var randomOffset = Math.floor(Math.random()\*500);

return 'https://api.github.com/users?since=' + randomOffset;

});

var startupRequestStream = Rx.Observable.just('https://api.github.com/users');

var requestStream = Rx.Observable.merge(

requestOnRefreshStream, startupRequestStream

);

There is an alternative and cleaner way of writing that, without the intermediate streams.

var requestStream = refreshClickStream

.map(function() {

var randomOffset = Math.floor(Math.random()\*500);

return 'https://api.github.com/users?since=' + randomOffset;

})

.merge(Rx.Observable.just('https://api.github.com/users'));

Even shorter, even more readable:

var requestStream = refreshClickStream

.map(function() {

var randomOffset = Math.floor(Math.random()\*500);

return 'https://api.github.com/users?since=' + randomOffset;

})

.startWith('https://api.github.com/users');

The [startWith()](https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md#rxobservableprototypestartwithscheduler-args) function does exactly what you think it does. No matter how your input stream looks like, the output stream resulting of startWith(x) will have x at the beginning. But I'm not [DRY](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself) enough, I'm repeating the API endpoint string. One way to fix this is by moving the startWith() close to the refreshClickStream, to essentially "emulate" a refresh click on startup.

var requestStream = refreshClickStream.startWith('startup click')

.map(function() {

var randomOffset = Math.floor(Math.random()\*500);

return 'https://api.github.com/users?since=' + randomOffset;

});

Nice. If you go back to the point where I "broke the automated tests", you should see that the only difference with this last approach is that I added the startWith().

## Modelling the 3 suggestions with streams

[](https://camo.githubusercontent.com/e581baffb3db3e4f749350326af32de8d5ba4363/687474703a2f2f692e696d6775722e636f6d2f4149696d5138432e6a7067)Until now, we have only touched a suggestion UI element on the rendering step that happens in the responseStream's subscribe(). Now with the refresh button, we have a problem: as soon as you click 'refresh', the current 3 suggestions are not cleared. New suggestions come in only after a response has arrived, but to make the UI look nice, we need to clean out the current suggestions when clicks happen on the refresh.

refreshClickStream.subscribe(function() {

// clear the 3 suggestion DOM elements

});

No, not so fast, pal. This is bad, because we now have **two** subscribers that affect the suggestion DOM elements (the other one being responseStream.subscribe()), and that doesn't really sound like [Separation of concerns](https://en.wikipedia.org/wiki/Separation_of_concerns). Remember the Reactive mantra?

So let's model a suggestion as a stream, where each emitted value is the JSON object containing the suggestion data. We will do this separately for each of the 3 suggestions. This is how the stream for suggestion #1 could look like:

var suggestion1Stream = responseStream

.map(function(listUsers) {

// get one random user from the list

return listUsers[Math.floor(Math.random()\*listUsers.length)];

});

The others, suggestion2Stream and suggestion3Stream can be simply copy pasted from suggestion1Stream. This is not DRY, but it will keep our example simple for this tutorial, plus I think it's a good exercise to think how to avoid repetition in this case.

Instead of having the rendering happen in responseStream's subscribe(), we do that here:

suggestion1Stream.subscribe(function(suggestion) {

// render the 1st suggestion to the DOM

});

Back to the "on refresh, clear the suggestions", we can simply map refresh clicks to null suggestion data, and include that in the suggestion1Stream, as such:

var suggestion1Stream = responseStream

.map(function(listUsers) {

// get one random user from the list

return listUsers[Math.floor(Math.random()\*listUsers.length)];

})

.merge(

refreshClickStream.map(function(){ return null; })

);

And when rendering, we interpret null as "no data", hence hiding its UI element.

suggestion1Stream.subscribe(function(suggestion) {

if (suggestion === null) {

// hide the first suggestion DOM element

}

else {

// show the first suggestion DOM element

// and render the data

}

});

The big picture is now:

refreshClickStream: ----------o--------o---->

requestStream: -r--------r--------r---->

responseStream: ----R---------R------R-->

suggestion1Stream: ----s-----N---s----N-s-->

suggestion2Stream: ----q-----N---q----N-q-->

suggestion3Stream: ----t-----N---t----N-t-->

Where N stands for null.

As a bonus, we can also render "empty" suggestions on startup. That is done by adding startWith(null) to the suggestion streams:

var suggestion1Stream = responseStream

.map(function(listUsers) {

// get one random user from the list

return listUsers[Math.floor(Math.random()\*listUsers.length)];

})

.merge(

refreshClickStream.map(function(){ return null; })

)

.startWith(null);

Which results in:

refreshClickStream: ----------o---------o---->

requestStream: -r--------r---------r---->

responseStream: ----R----------R------R-->

suggestion1Stream: -N--s-----N----s----N-s-->

suggestion2Stream: -N--q-----N----q----N-q-->

suggestion3Stream: -N--t-----N----t----N-t-->

## Closing a suggestion and using cached responses

There is one feature remaining to implement. Each suggestion should have its own 'x' button for closing it, and loading another in its place. At first thought, you could say it's enough to make a new request when any close button is clicked:

var close1Button = document.querySelector('.close1');

var close1ClickStream = Rx.Observable.fromEvent(close1Button, 'click');

// and the same for close2Button and close3Button

var requestStream = refreshClickStream.startWith('startup click')

.merge(close1ClickStream) // we added this

.map(function() {

var randomOffset = Math.floor(Math.random()\*500);

return 'https://api.github.com/users?since=' + randomOffset;

});

That does not work. It will close and reload all suggestions, rather than just only the one we clicked on. There are a couple of different ways of solving this, and to keep it interesting, we will solve it by reusing previous responses. The API's response page size is 100 users while we were using just 3 of those, so there is plenty of fresh data available. No need to request more.

Again, let's think in streams. When a 'close1' click event happens, we want to use the most recently emitted response on responseStream to get one random user from the list in the response. As such:

requestStream: --r--------------->

responseStream: ------R----------->

close1ClickStream: ------------c----->

suggestion1Stream: ------s-----s----->

In Rx\* there is a combinator function called [combineLatest](https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md#rxobservableprototypecombinelatestargs-resultselector) that seems to do what we need. It takes two streams A and B as inputs, and whenever either stream emits a value, combineLatest joins the two most recently emitted values a and b from both streams and outputs a value c = f(x,y), where f is a function you define. It is better explained with a diagram:

stream A: --a-----------e--------i-------->

stream B: -----b----c--------d-------q---->

vvvvvvvv combineLatest(f) vvvvvvv

----AB---AC--EC---ED--ID--IQ---->

where f is the uppercase function

We can apply combineLatest() on close1ClickStream and responseStream, so that whenever the close 1 button is clicked, we get the latest response emitted and produce a new value on suggestion1Stream. On the other hand, combineLatest() is symmetric: whenever a new response is emitted on responseStream, it will combine with the latest 'close 1' click to produce a new suggestion. That is interesting, because it allows us to simplify our previous code for suggestion1Stream, like this:

var suggestion1Stream = close1ClickStream

.combineLatest(responseStream,

function(click, listUsers) {

return listUsers[Math.floor(Math.random()\*listUsers.length)];

}

)

.merge(

refreshClickStream.map(function(){ return null; })

)

.startWith(null);

One piece is still missing in the puzzle. The combineLatest() uses the most recent of the two sources, but if one of those sources hasn't emitted anything yet, combineLatest() cannot produce a data event on the output stream. If you look at the ASCII diagram above, you will see that the output has nothing when the first stream emitted value a. Only when the second stream emitted value b could it produce an output value.

There are different ways of solving this, and we will stay with the simplest one, which is simulating a click to the 'close 1' button on startup:

var suggestion1Stream = close1ClickStream.startWith('startup click') // we added this

.combineLatest(responseStream,

function(click, listUsers) {l

return listUsers[Math.floor(Math.random()\*listUsers.length)];

}

)

.merge(

refreshClickStream.map(function(){ return null; })

)

.startWith(null);

## Wrapping up

And we're done. The complete code for all this was:

var refreshButton = document.querySelector('.refresh');

var refreshClickStream = Rx.Observable.fromEvent(refreshButton, 'click');

var closeButton1 = document.querySelector('.close1');

var close1ClickStream = Rx.Observable.fromEvent(closeButton1, 'click');

// and the same logic for close2 and close3

var requestStream = refreshClickStream.startWith('startup click')

.map(function() {

var randomOffset = Math.floor(Math.random()\*500);

return 'https://api.github.com/users?since=' + randomOffset;

});

var responseStream = requestStream

.flatMap(function (requestUrl) {

return Rx.Observable.fromPromise($.ajax({url: requestUrl}));

});

var suggestion1Stream = close1ClickStream.startWith('startup click')

.combineLatest(responseStream,

function(click, listUsers) {

return listUsers[Math.floor(Math.random()\*listUsers.length)];

}

)

.merge(

refreshClickStream.map(function(){ return null; })

)

.startWith(null);

// and the same logic for suggestion2Stream and suggestion3Stream

suggestion1Stream.subscribe(function(suggestion) {

if (suggestion === null) {

// hide the first suggestion DOM element

}

else {

// show the first suggestion DOM element

// and render the data

}

});

**You can see this working example at** [**http://jsfiddle.net/staltz/8jFJH/48/**](http://jsfiddle.net/staltz/8jFJH/48/)

That piece of code is small but dense: it features management of multiple events with proper separation of concerns, and even caching of responses. The functional style made the code look more declarative than imperative: we are not giving a sequence of instructions to execute, we are just **telling what something is** by defining relationships between streams. For instance, with Rx we told the computer that *suggestion1Stream* ***is*** the 'close 1' stream combined with one user from the latest response, besides being *null* when a refresh happens or program startup happened.

Notice also the impressive absence of control flow elements such as if, for, while, and the typical callback-based control flow that you expect from a JavaScript application. You can even get rid of the if and else in the subscribe() above by using filter() if you want (I'll leave the implementation details to you as an exercise). In Rx, we have stream functions such as map, filter, scan, merge, combineLatest, startWith, and many more to control the flow of an event-driven program. This toolset of functions gives you more power in less code.

## What comes next

If you think Rx\* will be your preferred library for Reactive Programming, take a while to get acquainted with the [big list of functions](https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md) for transforming, combining, and creating Observables. If you want to understand those functions in diagrams of streams, take a look at [RxJava's very useful documentation with marble diagrams](https://github.com/Netflix/RxJava/wiki/Creating-Observables). Whenever you get stuck trying to do something, draw those diagrams, think on them, look at the long list of functions, and think more. This workflow has been effective in my experience.

Once you start getting the hang of programming with Rx\*, it is absolutely required to understand the concept of [Cold vs Hot Observables](https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/creating.md#cold-vs-hot-observables). If you ignore this, it will come back and bite you brutally. You have been warned. Sharpen your skills further by learning real functional programming, and getting acquainted with issues such as side effects that affect Rx\*.

But Reactive Programming is not just Rx\*. There is [Bacon.js](http://baconjs.github.io/) which is intuitive to work with, without the quirks you sometimes encounter in Rx\*. The [Elm Language](http://elm-lang.org/) lives in its own category: it's a Functional Reactive Programming **language** that compiles to JavaScript + HTML + CSS, and features a [time travelling debugger](http://debug.elm-lang.org/). Pretty awesome.

Rx works great for event-heavy frontends and apps. But it is not just a client-side thing, it works great also in the backend and close to databases. In fact, [RxJava is a key component for enabling server-side concurrency in Netflix's API](http://techblog.netflix.com/2013/02/rxjava-netflix-api.html). Rx is not a framework restricted to one specific type of application or language. It really is a paradigm that you can use when programming any event-driven software.

If this tutorial helped you, [tweet it forward](https://twitter.com/intent/tweet?original_referer=https%3A%2F%2Fgist.github.com%2Fstaltz%2F868e7e9bc2a7b8c1f754%2F&text=The%20introduction%20to%20Reactive%20Programming%20you%27ve%20been%20missing&tw_p=tweetbutton&url=https%3A%2F%2Fgist.github.com%2Fstaltz%2F868e7e9bc2a7b8c1f754&via=andrestaltz).

### Legal

© Andre Cesar de Souza Medeiros (alias "Andre Staltz"), 2014. Unauthorized use and/or duplication of this material without express and written permission from this site’s author and/or owner is strictly prohibited. Excerpts and links may be used, provided that full and clear credit is given to Andre Medeiros and <http://andre.staltz.com> with appropriate and specific direction to the original content.

[Creative Commons License](http://creativecommons.org/licenses/by-nc/4.0/)  
"Introduction to Reactive Programming you've been missing" by [Andre Staltz](http://andre.staltz.com) is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](http://creativecommons.org/licenses/by-nc/4.0/).  
Based on a work at <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>.

# [Reactive log stream processing with RxJava - Part I](https://balamaci.ro/reactive-log-processing/)

### Centralized logs as a data source for realtime data analysis

In the previous blog entries we saw how we can leverage the power of the ELK stack for log collection and analysis of our Java apps.

With the move towards microservices or containerization of applications it becomes the defacto standard to have a stack for centralized log processing and storage.   
Can we maybe go the next step and use that information proactively instead of just for just finding the cause of problems long after they appeared?

If **we were to consider the log events as a stream of data for things happening in realtime** in your system it would be very interesting to tap into and perform **realtime data analysis** with all sorts of uses like **detecting fraudulent behavior** for example by aggregating different streams of information while the "attack" is happening and block the attacker instead of the "traditional" way of just using log data for forensics to investigate after the issue appeared.

For ex. we could **filter** only events of a certain type, **'group by'** a common key like the userID and count them in a time-window to get the number of actions of that type the user is doing in a certain timeframe.

failedLoginsStream()

.window(5, TimeUnit.SECONDS)

.flatMap(window ->

window

.groupBy(propertyStringValue("remoteIP"))

.flatMap(grouped -> grouped

.count()

.map(failedLoginsCount -> {

final String remoteIp = grouped.getKey();

return new Pair<>(remoteIp, failedLoginsCount);

}))

)

.filter(pair -> pair.get > 10)

.forEach(system.out:println);

We could trigger queries in other systems and treat those responses as streams to which we can subscribe and apply **a multitude of common stream operators** that reactive streams frameworks provide.

And the good thing is that since you're working with already existing log outputs you can build this separate and extend it without touching or polluting the business logic of the application.

### Learning a new programming paradigm

This could be a good excuse to get into the world of **Reactive stream programming** and we don't really need something as big as [Kafka Streams](http://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple) [Spark](https://spark.apache.org/docs/latest/streaming-programming-guide.html) or **Flink** to get us started.   
Reactive programming is about **non-blocking**, **event-driven** applications that scale even on a small number of threads with back-pressure(feedback mechanism to ensure producers do not overwhelm consumers).

The biggest new thing **Spring5 will bring will be** [**Reactive support**](https://spring.io/blog/2016/04/19/understanding-reactive-types). A new module [spring-web-reactive](https://github.com/spring-projects/spring-framework/tree/master/spring-web-reactive) a framework similar with spring-web-mvc that enables async response(non blocking) REST services and a reactive web-client and that will probably work great for microservice architectures.   
The reactive-streams concepts is not just Spring specific, but instead there is a common specification [reactive-streams-jvm](https://github.com/reactive-streams/reactive-streams-jvm) agreed upon by the major reactive frameworks(so while there might not be exact name matches the concepts will make it easy to switch frameworks).   
Historically Rx.NET introduced the reactive-streams model, and Netflix ported it to java with RxJava. Then the concept has been implemented into other languages as well, under the [Reactive EXtensions](http://reactivex.io/) umbrella.   
Then since other companies were going kinda in the same direction the [reactive-streams](http://www.reactive-streams.org/) specification took of. Now **RxJava** since it was kind of the pioneer needs to do a bigger refactor(in version [**2.x**](https://github.com/ReactiveX/RxJava/wiki/What%27s-different-in-2.0) to better match the specification, while **Spring reactor** being newer could start fresh with directly implementing the spec.   
You care read more about how they [relate](https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape).

Also reactive streams will come to Java 9 as Doug Lea wants to include the reactive-streams under container object [java.util.concurrent.Flow](http://gee.cs.oswego.edu/dl/jsr166/dist/docs/java/util/concurrent/Flow.html)

#### Benefits from a performance perspective

Also another buzzword right now is the **microservices architecture** where you need to be able to make requests to multiple other services. Ideally you'd want to do it **without blocking** waiting for the whole response before making the request to the next service. Think that instead of waiting for a whole possibly huge List to be returned by a service, it might be the case to start a query in another system as soon as the first element is available.

Treating that remote call response as a Stream to which we subscribe for an action when the response arrives, instead of blocking the thread waiting for the response means we can use **less threads overall** which in turn means **less resources wasted(cpu for context switching between threads and memory for each thread stack)**.   
So by using reactive programming we should be **able to handle a larger amount of log events with commodity hardware**.

An example: If we're a service like GMail and we need to display the user's emails. However emails in turn might have many people in CC. It would be nice to display a photo of those that the user has in his contacts - which means a REST call in the ContactsService

We'd normally have something like

Future<List<Mail>> emailsFuture = mailstoreService.getUnreadEmails();

List<Mail> emails = emailsFuture.get(); //blocking the current thread

//waiting possibly for a long time to get the whole list

//we cannot start next processing as soon as the first email is found??

Future<List<Contacts>> contacts = getContactsForEmails(emails);

for(Mail mail : emails) {

streamRenderEmails(mail, contacts); //push emails to the client

}

Partially the problem has been improved with the reactive support from Java8 with **CompletableFuture**(which with it's thenCompose, thenCombine, thenAccept and other 50 something methods it's not making it easy to remember what each method does, which in turn I think doesn't help readability).

CompletableFuture<List<Mail>> emailsFuture = mailstoreService.getUnreadEmails();

CompletableFuture<List<Contact>> emailsFuture

.thenCompose(emails -> getContactsForEmails(emails)) //we still need to wait for the List<Mail> to

.thenAccept(emailsContactsPair -> streamRenderEmails(emailsContactsPair.getKey(), emailsContactsPair.getValue()))

we could switch to an Iterator data type instead of List still there are no methods to tell it to do something when a new value arrives. SQL does this with returning the ResultSet(on which you can do rs.next()) instead of getting the whole data in memory.

public interface Iterator<E> {

/\*\*

\* Returns {@code true} if the iteration has more elements.

\*/

boolean hasNext();

/\*\*

\* Returns the next element in the iteration.

\*/

E next();

}

But we need to constantly ask "do you have a new value"?

Iterable<Mail> emails = mailstoreService.getUnreadEmails();

Iterator<Mail> emailsIt = emails.iterator();

while(emailsIt.hasNext()) {

Mail mail = emailsIt.next(); //nonblockin but we still need to constantly waste cpu asking

if(mail != null) { //for new values

....

}

}

What we'd need instead would be like a **reactive Iterator**, a datatype to which you could subscribe for an action to be executed once there is a new value ready and this is where reactive stream programming begins.

#### So what is a Stream?

A Stream is simply **a sequence of events ordered in time** (eventX was emitted after eventY, **there are no concurrent events**).

A Stream is modeled so it can emit **0..N events** and **either one of two terminal operations**:   
- **completed** event through which it signals subscribers that it finished emitting data   
- **error** event for signaling it finished exceptionally

We can describe that visually with the use of 'marble diagrams'.

So everything can be thought as being a Stream, not just log events. Even a single value can be expressed as a Stream by emitting the value followed by an **completed** event.   
An infinite stream is one that only emits events but not any of the terminal events(completed | error).

RxJava defines the **Observable** **data type** to model the Stream of events of type <T>. Spring Reactor's equivalent is [Flux](https://projectreactor.io/core/docs/api/reactor/core/publisher/Flux.html)

* Observable<Double> to represent a stream of temperatures taken at various intervals.
* Observable<CartItem> to represent a stream of products bought in our web store.
* Observable<User> to represent a single User returned by a DB query

public Observable<User> findByUserId(String userId) {...}

//or Single for being more explicit

public Single<User> findByUserId(String userId) {...}

But **Observable<T>** is just a datatype and same as the Publish/Subscriber pattern, we need a Subscriber to process the 3 types of events

Observable<CartItem> cartItemsStream = ...;

Subscriber<CartItem> subscriber = new Subscriber<CartItem>() {

@Override

public void onNext(CartItem cartItem) {

System.out.println("Cart Item added " + cartItem);

}

@Override

public void onCompleted() {

}

@Override

public void onError(Throwable e) {

e.printStackTrace();

}

};

cartItemsStream.subscribe(subscriber);

#### Reactive operators

But that is just the Stream part, and until now we've not been doing something more special than the classic Observer pattern.   
The **Reactive** part means we can **define some Function**(the **operators**) that will be executed when the stream emits an event.   
That means another stream will be created(streams are immutable) to which we can subscribe another operator and so on.

Observable<CartItem> filteredCartStream = cartStream.filter(new Func1<CartItem, Boolean>() {

@Override

public Boolean call(CartItem cartItem) {

return cartItem.isLaptop();

}

});

Observable<Long> laptopCartItemsPriceStream = filteredCartStream.map(new Func1<CartItem, Long>() {

@Override

public Long call(CartItem cartItem) {

try {

return priceService.getPrice(cartItem.getId());

} catch(PriceServiceException e) {

thrown new RuntimeException(e);

}

}

});

Since the operator methods of the Observable class(filter, map, groupBy,...) return Observable, it means **we can chain the operators together** and combined with lambda syntax we can write something pretty

Observable<BigDecimal> priceStream = cartStream

.filter((cartItem) -> cartItem.isLaptop()).

.map((laptop) -> {

try {

return priceService.getPrice(cartItem.getId());

} catch(PriceServiceException e) {

thrown new RuntimeException(e);

}

});

The thing to notice above is that **when creating priceStream nothing is happening** in the sense that **priceService.getPrice()** is not getting invoked until there is an item flowing through the operators chain.   
That means we created through the rx-operators sort of a blueprint of how the data will be manipulated once it starts flowing downstream(a subscriber registers).

When asked to explain **reactive programming** people jokingly give as an example an Excel sheet where you write the formulas for columns and as soon as you update a cell the formula is triggered which updates another cell which in turn triggers another formula and so on. Just like that the rx-operators don't do anything by themselves they are just formulas for data manipulation and each gets a chance to do it's thing before passing it down the chain.

To help understand better how events travel along the operators chain I found helpful the analogy of the house movers proposed by [Thomas Nield](https://tomstechnicalblog.blogspot.ro/2015/10/understanding-observables.html) where each house mover is an operator passing along house objects.

He gives as example the following code:

Observable<Item> mover1 = Observable.create(s -> {

while (house.hasItems()) {

s.onNext(house.getItem());

}

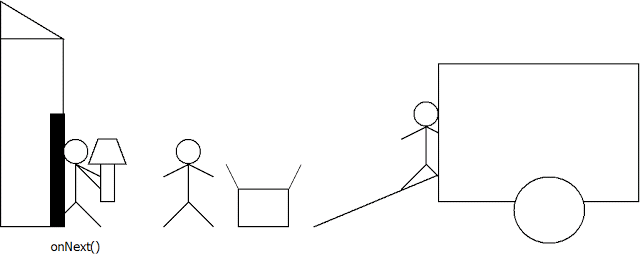
s.onCompleted();

});

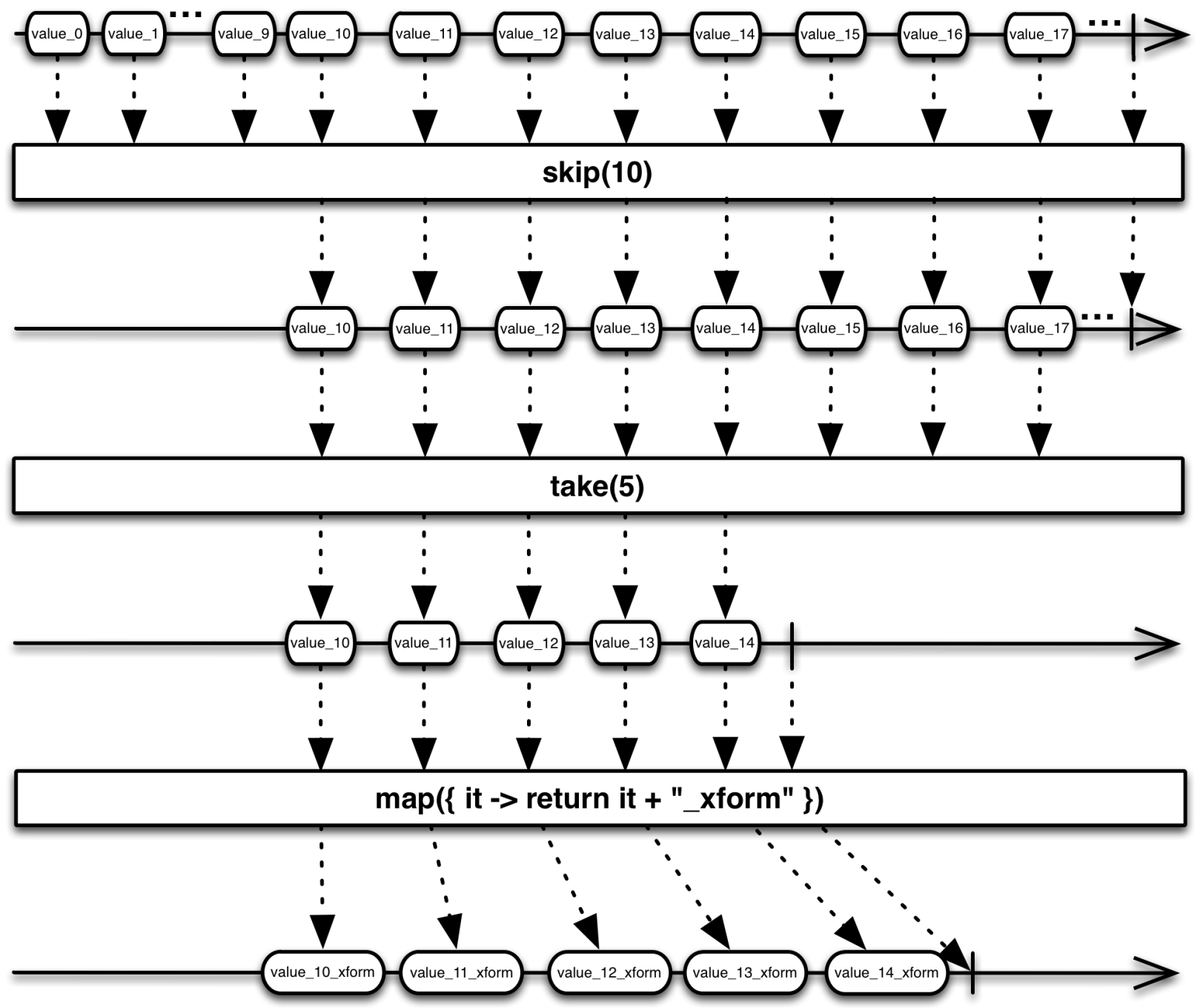
Observable<Item> mover2 = mover1.map(item -> putInBox(item));

Subscription mover3 = mover2.subscribe(box -> putInTruck(box),

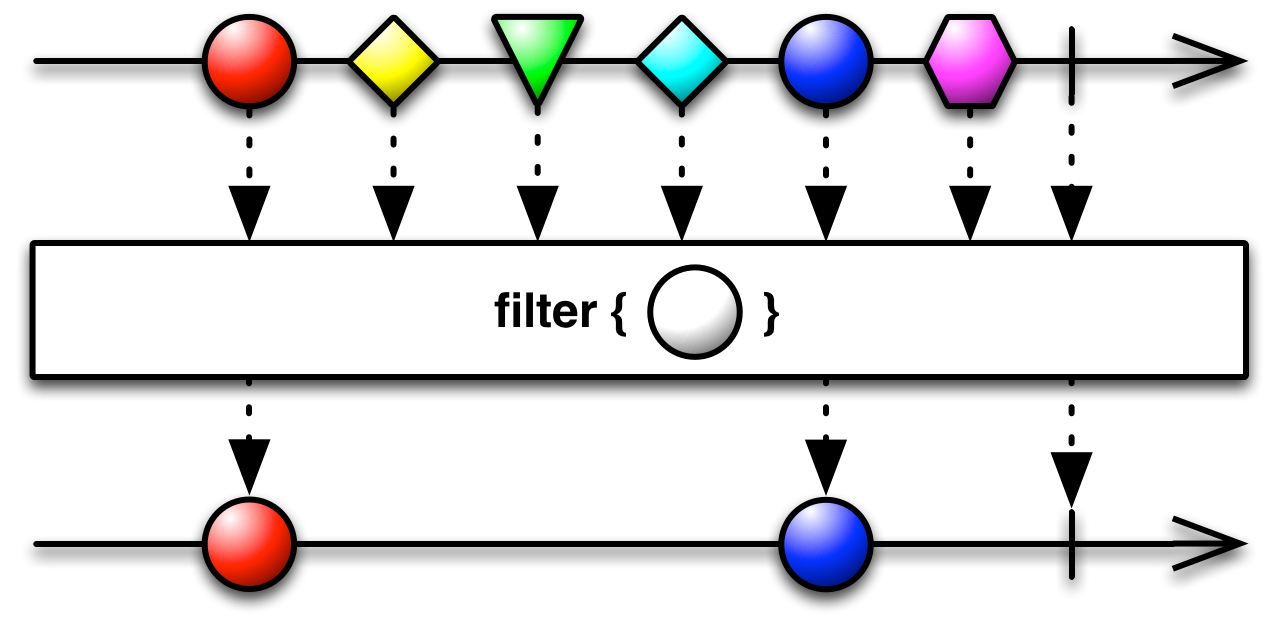
() -> closeTruck()); //this is what runs for onCompleted



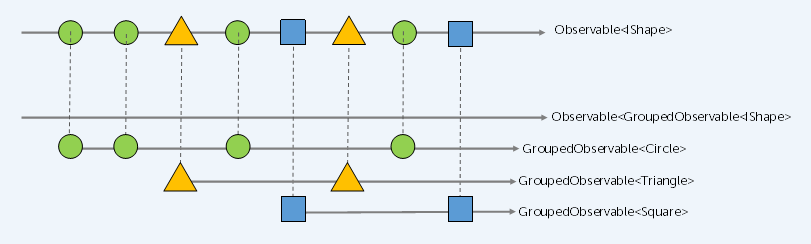
"**Mover 1** on the far left is the **source Observable**. He creates the emissions by picking items out of the house. He then calls onNext() on **Mover 2, who does a map() operation**. When his onNext() is called he takes that Item and puts it in a Box. Then he calls onNext() on **Mover 3, the final Subscriber** who loads the box on the truck."

The magic or RxJava is the large set of operators available and your job on how to combining them together to manipulate the flow of data.   


The **many Stream operators** help establish **a common vocabulary of manipulations when dealing with streams** that can have implementations in popular languages(RxJava, RxJS, Rx.NET, etc) of the ReactiveX framework.   
These concepts should be familiar even when using different reactive streams frameworks like [Spring Reactor](https://balamaci.ro/reactive-log-processing/)(with the hope of an agreement for a common base of [operators](https://github.com/reactor/reactive-streams-commons)).

Until now we only saw simple operators like **filter**:   
  
Which means but it only pushes downstream elements which pass the filtering condition(one mover drops everything with a value < 100$ instead of passing it to the next mover)

There are however operators which can split a stream into many streams - Observable<Observable<T>>(Stream of Streams) - operators like **groupBy**



Observable<Integer> values = Observable.just(1,4,5,7,8,9,10);

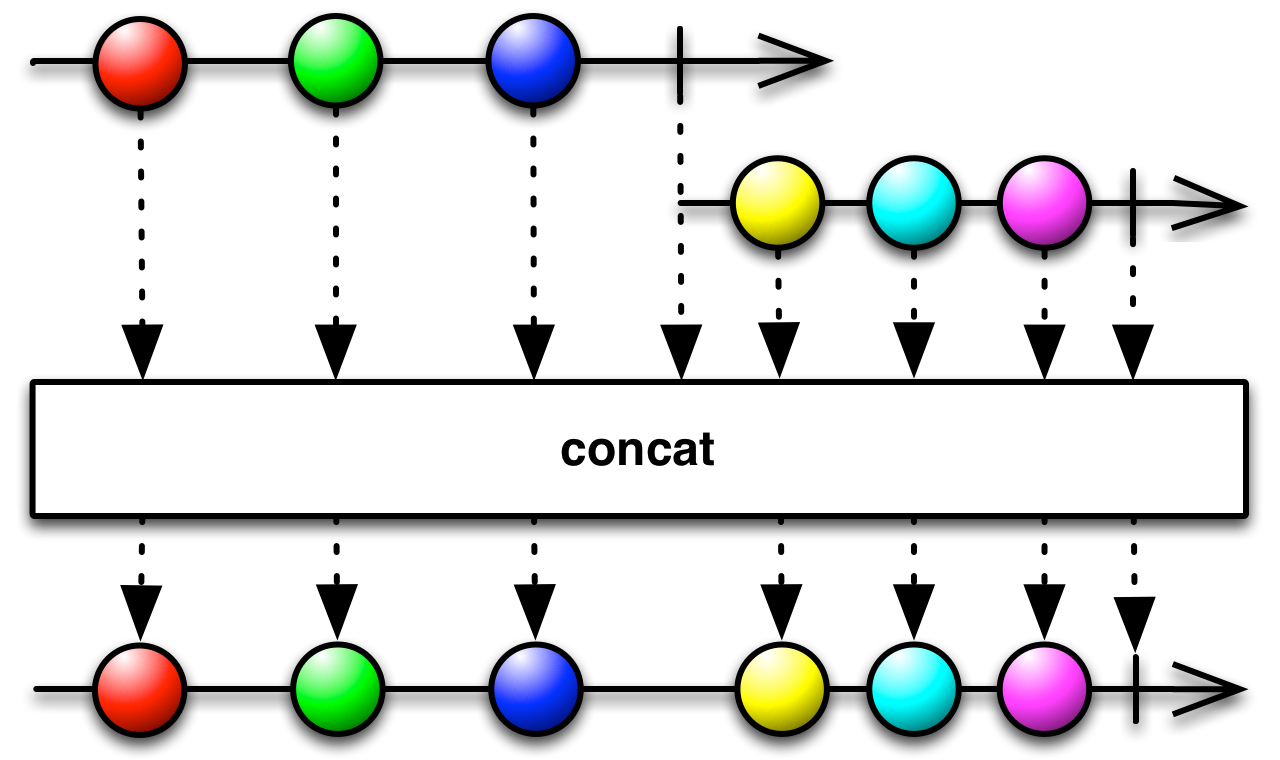
Observable<GroupedObservable<String, Integer>> oddEvenStream = values.groupBy((number) -> number % 2 == 0 ? "odd":"even");

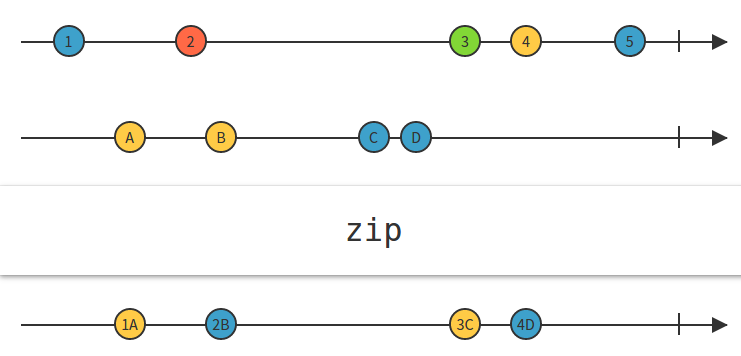
Observable<Integer> remergedStream = Observable.concat(oddEvenStream);

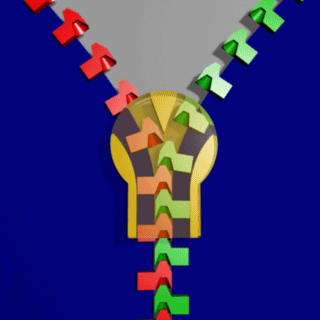
remergedStream.subscribe(number -> System.out.print(number +" "));

//Outputs

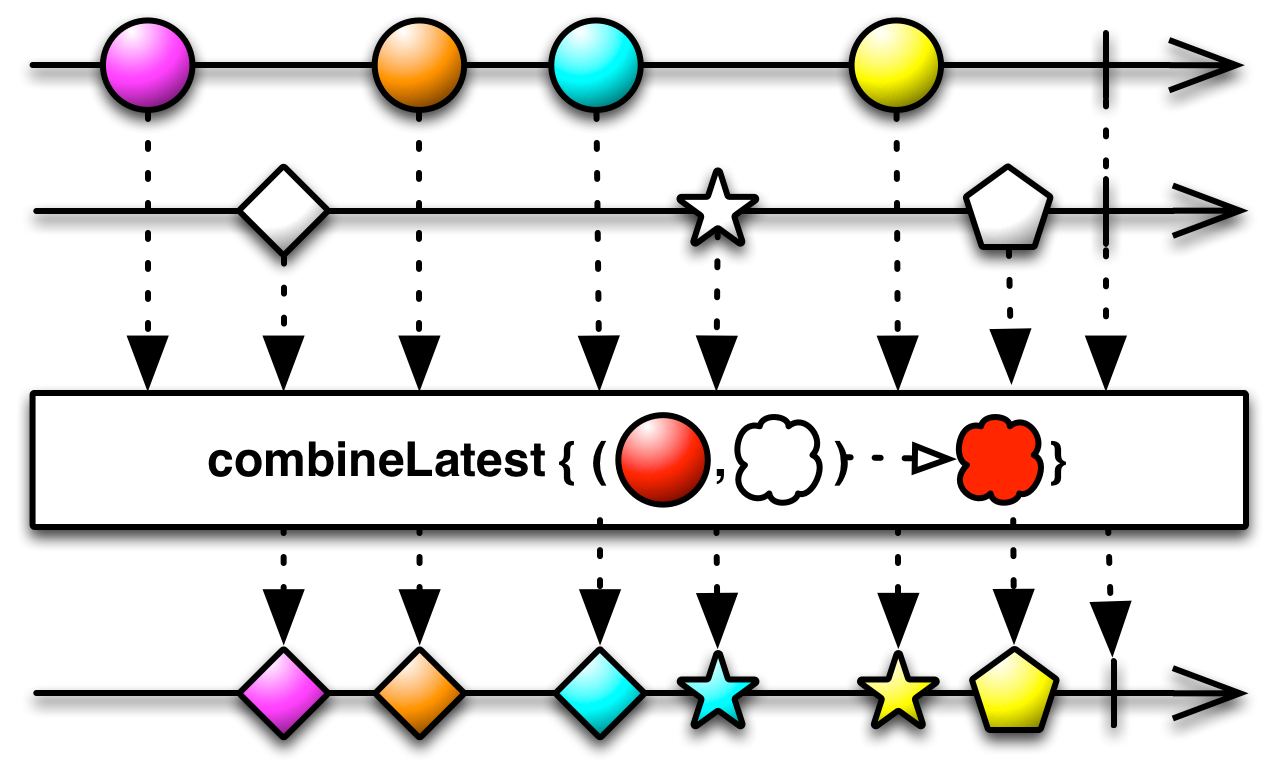
//1 5 7 9 4 8 10

and the rather simple operator **concat** which merges back the "odd" and "even" stream into a single one to which we subscribe.   
  
as we can see the **concat** operator waits for a stream to complete before appending another one and so on, creating back a single stream. Thus the odd numbers were displayed first.

Also we have the way to merge back together multiple streams like **zip** operator   


Zip is not named that way because it's acting like an archiving program but rather from the way a **zipper** works to combine events from two streams.   
  
It's taking one event from a stream and pairs it with another from the other stream as soon as one is ready, and applies a merging operator before sending it downstream.   
PS: It also works with more than just two streams.

So even if one stream is emitting faster, the downstream listener will only see the combined event as soon as there is a matching event being emitted on the slower stream.   
It's actually very useful as a way to "wait" for the response of multiple remote calls which return streams.

The **combineLatest** on the other hand it's not waiting for a pair emission to appear but instead uses the last emission from the slower stream before applying the merge function and sending it downstream.   


#### Moving to a Push based mindset

Let's see some examples of how you can actually create Observables. The most verbose variant but which let us understands more:

log.info("Before create Observable");

Observable<Integer> someIntStream = Observable

.create(new Observable.OnSubscribe<Integer>() {

@Override

public void call(Subscriber<? super Integer> subscriber) {

log("Create");

subscriber.onNext(3);

subscriber.onNext(4);

subscriber.onNext(5);

subscriber.onCompleted();

log("Completed");

}

});

log.info("After create Observable");

log.info("Subscribing 1st");

someIntStream.subscribe((val) -> log.info("received " + val)); //we don't have to implement

//the other methods(for onError and onComplete) if we don't want to do something specific

log.info("Subscribing 2nd");

someIntStream.subscribe((val) -> log.info("received " + val));

**Events are pushed onto the subscriber as soon as it subscribes**. It's not doing this on construction, here we just passed it a **new OnSubscribe** object which represent what to do when someone subscribes.   
Until we subscribe to the Observable there is no output, there is nothing happening - the data is not flowing-.   
When someone subscribes, the call() method is invoked and 3 events are pushed downstream followed by the signal that the stream completed.

Above we subscribed twice, the code inside call(...) method will be invoke twice also. So it's effectively re-pushing the same values as soon as someone else subscribes and the following output will be produced:

mainThread: Before create Observable

mainThread: After create Observable

mainThread: Subscribing 1st

mainThread: Create

mainThread: received 3

mainThread: received 4

mainThread: received 5

mainThread: Completed

mainThread: Subscribing 2nd

mainThread: Create

mainThread: received 3

mainThread: received 4

mainThread: received 5

mainThread: Completed

Important thing to notice is that rx operators don't necessarily mean multithreading. RxJava doesn't inject any concurrency by default between the Observable and the Subscriber. This is why all the calls are happening on the 'main' thread.

This kind of Observable that is starting even emission when someone subscribes is what we call **cold observables**. The other type is **hot observables**, they emit events even when nobody is subscribed.

* **Cold Observables** Only begins emitting the events when someone subscribes - they start the work(ex: makes a DB query) when subscribed to-. Each subscriber gets the same events. Sort of **like a CD where the same songs are played** to whomever puts the cd into the player to listen.
* **Hot Observables** Events are emitted even when there are no subscribers. **Like a radio stations where it plays the song in the air even when nobody is tuned in**. Just like when you tune in later on a station, you miss previous events. Hot observables model events to which you don't have control over when they emit. Like when the log events are produced.

**Subject** s are special kind of Observable that is also an Observer(like Subscriber - which means we can push events(call onNext()) to them-) and make implementing hot Observables easier. There exists more implementations like **ReplaySubject** that keeps the emitted events in a buffer and replay them on subscription(you can ofcourse specify the size of the buffer to prevent OutOfMemory), while **PublishSubject** only pass on events that happened after subscription.

And of course there are more static helper methods for creating Observables from other sources

Observable.just("This", "is", "something")

Observable.from(Iterable<T> collection)

Observable.from(Future<T> future) - emits the value when the future completes

### Adding a push based data emitter to our ELK stack - RabbitMQ

In a traditional [ELK](https://balamaci.ro/java-app-monitoring-with-elk-logstash/) stack we use ElasticSearch to query the log events data so a more **'pull based'** system.   
Could we have instead a **push based** where we're notified **"immediately"** when another log event appears to further reduce the reaction time from when the event is produced to when we begin processing it.   
One of the many possible solutions would be **RabbitMQ** as being a battle tested solution with a very good reputation for performance for handling a huge amount of messages. Besides that Logstash already supplies a plugin for RabbitMQ(there is also one for FluentD) so we can easily integrate it in our existing ELK stack and write the log data both to ElasticSearch and RabbitMQ.   
You may remember from that Logstash can act as a controller and choose how to process and where to send/store the log events. That means we can decide to filter log events we want to process or to send them to different RabbitMQ queues.

There is even the option to send data directly to RabbitMQ through a Logback [Appender](http://docs.spring.io/spring-amqp/api/org/springframework/amqp/rabbit/logback/AmqpAppender.html) should you want to bypass Logstash.   
Sidenote: While named AmqpAppender, it's rather specific to RabbitMQ AMQP implementation(AMQP protocol version 0-9-1, 0-9). ActiveMQ for ex.(while also supplying an AMQP connector) seems to implement AMQP protocol 1.0, while the spring-amqp library works with the protocol versions 0-9-1, 0-9 which are totally different at the wire level than 1.0) so you'll encounter a nice exception like   
'org.apache.activemq.transport.amqp.AmqpProtocolException: Connection from client using unsupported AMQP attempted'

However our solution was to use [logstash-logback-encoder](https://github.com/logstash/logstash-logback-encoder) to send JSON formatted log events to Logstash. We'll now redirect the logstash output to a RabbitMQ exchange.

We'll use **docker-compose** to start up a logstash-rabbitmq cluster. You can clone the [repo](https://github.com/balamaci/blog-elk-docker).

docker-compose -f docker-compose-rabbitmq.yml up

and you can use **./event-generate.sh** to generate some random events that get pushed to logstash.

It's using the logstash configuration [file](https://github.com/balamaci/blog-elk-docker/blob/rabbitmq/logstash/config-rabbitmq/logstash.conf) to specify where we want to send the data. We use the [rabbitmq-output-plugin](https://www.elastic.co/guide/en/logstash/current/plugins-outputs-rabbitmq.html) as reference:

output {

rabbitmq {

exchange => logstash

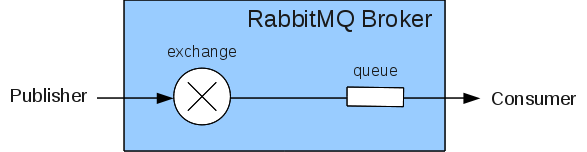
exchange\_type => direct

host => rabbitmq

key => my\_app

}

}

RabbitMQ is not a traditional JMS server, instead it uses the **AMQP protocol** which has quite a different concept for queues.   
  
A publisher sends messages to a named **exchange** and a consumer pulls messages from a queue. The message has a standard header 'routing-key' which is used in a process called **binding** to associate an exchange message to a queue. The queues can filter which messages they receive via the binding key and you can use wildcards in the binding like 'logstash.\*'

For an indepth explanation for AMQP you can read [here](https://spring.io/blog/2010/06/14/understanding-amqp-the-protocol-used-by-rabbitmq/) and [here](https://www.cloudamqp.com/blog/2015-09-03-part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html).

So we [configured](https://github.com/balamaci/rxjava-rabbitmq/blob/master/src/main/java/com/balamaci/rx/configuration/AmqpSourceConfiguration.java) a Spring connection to RabbitMq

@Bean

ConnectionFactory connectionFactory() {

return new CachingConnectionFactory(host, port);

}

@Bean

RabbitAdmin rabbitAdmin() {

RabbitAdmin rabbitAdmin = new RabbitAdmin(connectionFactory());

rabbitAdmin.declareQueue(queue());

rabbitAdmin.declareBinding(bindQueueFromExchange(queue(), exchange()));

return rabbitAdmin;

}

@Bean

SimpleMessageListenerContainer container(ConnectionFactory connectionFactory, MessageListenerAdapter listenerAdapter) {

SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();

container.setConnectionFactory(connectionFactory);

container.setQueueNames(queueName);

container.setMessageListener(listenerAdapter);

return container;

}

@Bean

Queue queue() {

return new Queue(queueName, false);

}

DirectExchange exchange() {

return new DirectExchange("logstash");

}

private Binding bindQueueFromExchange(Queue queue, DirectExchange exchange) {

return BindingBuilder.bind(queue).to(exchange).with("my\_app");

}

@Bean

MessageListenerAdapter listenerAdapter(Receiver receiver) {

MessageListenerAdapter messageListenerAdapter = new MessageListenerAdapter(receiver,

new MessageConverter() {

public Message toMessage(Object o, MessageProperties messageProperties)

throws MessageConversionException {

throw new RuntimeException("Unsupported");

}

public String fromMessage(Message message) throws MessageConversionException {

try {

return new String(message.getBody(), "UTF-8");

} catch (UnsupportedEncodingException e) {

throw new RuntimeException("UnsupportedEncodingException");

}

}

});

messageListenerAdapter.setDefaultListenerMethod("receive"); //the method in our Receiver class

return messageListenerAdapter;

}

@Bean

Receiver receiver() {

return new Receiver();

}

We defined a queue and bind it to the 'logstash' exchange to receive messages with the 'my\_app' routing key.   
The **MessageListenerAdapter** above defines that the 'receive' method should be called on Receiver bean every time a new message is received from the queue.

Since we're expecting a continuous stream of log events we don't have control over, we can think of using a hot observable that pushes events to all subscribers after they subscribed so we can use **PublishSubject** for the job.

public class Receiver {

private PublishSubject<JsonObject> publishSubject = PublishSubject.create();

public Receiver() {

}

/\*\*

\* Method invoked by Spring whenever a new message arrives

\* @param message amqp message

\*/

public void receive(Object message) {

log.info("Received remote message {}", message);

JsonElement remoteJsonElement = gson.fromJson ((String) message, JsonElement.class);

JsonObject jsonObj = remoteJsonElement.getAsJsonObject();

publishSubject.onNext(jsonObj);

}

public PublishSubject<JsonObject> getPublishSubject() {

return publishSubject;

}

}

We need to be aware that event the **SimpleMessageListenerContainer** supports having more than one thread that consumes from the queue(and emits the events downstream). However the Observable contract says we cannot emit events concurrently(calls to onNext,onComplete,onError must be serialized):

// DO NOT DO THIS

Observable.create(s -> {

// Thread A

new Thread(() -> {

s.onNext("one");

s.onNext("two");

}).start();

// Thread B

new Thread(() -> {

s.onNext("three");

s.onNext("four");

}).start();

});

// DO NOT DO THIS

//DO THIS

Observable<String> obs1 = Observable.create(s -> {

// Thread A

new Thread(() -> {

s.onNext("one");

s.onNext("two");

}).start();

});

Observable<String> obs2 = Observable.create(s -> {

// Thread B

new Thread(() -> {

s.onNext("three");

s.onNext("four");

}).start();

});

Observable<String> c = Observable.merge(obs1, obs2);

We could go around this problem by calling Observable.serialize() or Subject.toSerialized() but since we just go with the default of 1 Thread in the ListenerContainer, there is no need to do so. Still you need to be aware of this if you plan to use Subjects as an event bus pushing events onto from multiple threads. Read a more [indepth explanation](https://artemzin.com/blog/rxjava-thread-safety-of-operators-and-subjects/).

For now, you can checkout out the code from the [repo](https://github.com/balamaci/rxjava-rabbitmq) as we continue this long post in [Part II](https://balamaci.ro/reactive-log-stream-processing-with-rxjava-part-2/) or go to the [Rx Playground](https://github.com/balamaci/rxjava-playground) for some **more examples**.

# [Reactive log stream processing with RxJava - Part II](https://balamaci.ro/reactive-log-stream-processing-with-rxjava-part-2/)

In the previous post we saw how we can add a push based solution(RabbitMQ) to our "ELK" stack and how we can connect from Spring to RabbitMQ and have the log events emitted as a reactive stream.

#### Json file as the source of log events

Since our main target is to play around as easily as possible with RxJava operators, we'll simulate receiving the events from RabbitMQ by reading them from a json file instead.   
This helps that we don't need to start the full stack of RabbitMQ+Logstash+Log emitter app(through docker-compose) to test our operator chain setup, and also having the same events being emitted in a deterministic manner helps us track down where we went wrong.

We'll be using Spring Profiles to switch between how the events are been generated so apart from the [AmqpSourceEmitterConfiguration](https://github.com/balamaci/blog-rxjava-rabbitmq/blob/master/src/main/java/com/balamaci/rx/configuration/AmqpSourceEmitterConfiguration.java) we'll have [JsonFileEmitterSourceConfiguration](https://github.com/balamaci/blog-rxjava-rabbitmq/blob/master/src/main/java/com/balamaci/rx/configuration/JsonFileEmitterSourceConfiguration.java) which pushes the json entries onto the **PublishSubject**(to be consistent with how the AMQP MessageListenerAdapter does it).

@Configuration

@Profile("hardcoded-events")

public class JsonFileEmitterSourceConfiguration implements ApplicationListener<ContextRefreshedEvent> {

@Bean

public Receiver receiver() {

return new Receiver();

}

@Bean(name = "events")

public Observable<JsonObject> emitEvents(Receiver receiver) {

return receiver.getPublishSubject();

}

@Override

public void onApplicationEvent(ContextRefreshedEvent event) {

Receiver receiver = event.getApplicationContext().getBean(Receiver.class);

startEmitting(receiver);

}

private void startEmitting(Receiver receiver) {

PublishSubject<JsonObject> publishSubject = receiver.getPublishSubject();

Supplier<Integer> waitForMillis = () -> 200; //force a little fixed delay between the events emitter.

Executors.newSingleThreadExecutor() //we start a single separate thread to simulate the same conditions like

.submit(() -> produceEventsFromJsonFile(publishSubject, waitForMillis));

}

private void produceEventsFromJsonFile(Observer<JsonObject> subscriber, Supplier<Integer> waitTimeMillis) {

JsonArray events = Json.readJsonArrayFromFile("events.json");

events.forEach(ev -> {

sleepMillis(waitTimeMillis.get());

JsonObject jsonObject = (JsonObject) ev;

log.info("Emitting {}", Json.propertyStringValue("message").call(jsonObject));

subscriber.onNext(jsonObject);

});

}

}

so now we have a Spring Bean named "events" which is an Observable - the main stream of events -.

and we can switch between profiles at start time

public static void main(String[] args) throws Exception {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();

context.setDisplayName("RxJava");

context.getEnvironment().setActiveProfiles("hardcoded-events");

//context.getEnvironment().setActiveProfiles("amqp");

context.scan("com.balamaci.rx.configuration");

context.scan("com.balamaci.rx.observable");

context.refresh();

context.start();

}

Until this point even if events are being emitted(since we're using a hot observable), nobody is yet subscribed so the events are just lost.

Let's start by just displaying the failed login events and in the end show how many failed logins there were from each ip.

The events being emitted are just some simulated Login events that we load from [events.json](https://github.com/balamaci/rxjava-rabbitmq/blob/master/events.json) which look like:

{

"appName": "elk-testdata",

"level": "INFO",

"logger\_name": "ro.fortsoft.elk.testdata.generator.event.LoginEvent",

"thread\_name": "pool-3-thread-4",

"message": "SUCCESS login for user\u003d\u0027giannaodom@yahoo.com\u0027",

"remoteIP": "192.168.0.32",

"userName":"giannaodom@yahoo.com",

"@timestamp": "2016-06-30T11:33:40.872Z",

"host": "172.17.0.4"

},

{

"appName": "elk-testdata",

"level": "ERROR",

"logger\_name": "ro.fortsoft.elk.testdata.generator.event.LoginEvent",

"thread\_name": "pool-3-thread-6",

"message": "FAILED login for user\u003d\u0027perez@yahoo.com\u0027",

"remoteIP": "192.168.0.105",

"userName":"perez@yahoo.com",

"@timestamp": "2016-06-30T11:33:40.874Z",

"host": "172.17.0.4",

}

@Configuration

public class LoginObservables {

@Autowired

@Qualifier("events")

protected Observable<JsonObject> events;

@Bean

Observable<JsonObject> loginEvents() {

return events

.filter(checkPropertyFunc("logger\_name", val -> val.contains("LoginEvent")));

}

@Bean

Observable<JsonObject> failedLogins() {

return loginEvents()

.filter(Json.checkPropertyFunc("message", val -> val.startsWith("FAILED login")));

.doOnNext((jsonObject) -> log.info("Debug {}", Json.propertyStringValue("messsage")));

}

@Bean

public Subscription displayLogEventsSubscription() {

failedLogins()

.subscribe((jsonObject) -> log.info("Got {}", Json.propertyStringValue("messsage")));

}

}

Remember until we connect the "sink"(the subscriber) events are not traveling down the operators chain -the operator functions are not executed-.   
We can prove this by adding **'onNext()'** - which is just a simple way to interpose an action in the chain, great for debugging purposes.   
We registered the subscriber by annotating @Bean the 'displayLogEventsSubscription' that method is invoked automatically by Spring.   
Without the Subscription bean the 'Debug' message is not seen.

### Too many failed logins from the same ips

Now let's dive right in and try to see the cases where there are too many failed logins from a certain ip.   
First we need a time range to measure the failed logins so we get a result of X failed logins / 10 secs.   
We can use the **window** operator for this.

Operator **window(long timespan, TimeUnit unit)** splits the events stream into multiple windows of fixed time. It returns a stream of stream(Observable<Observable<T>>) each stream emitting events in it's timerange.

Observable<Observable<JsonObject>> windowStream = failedLogins().window(30, TimeUnit.SECONDS);

Sidenote: You might wonder why not a single continuous stream as return value? Well, because that the window operator needs to signal **onComplete** after it's reached the specified time value, and the reactive streams specs say that you cannot emit new events after **onComplete**.

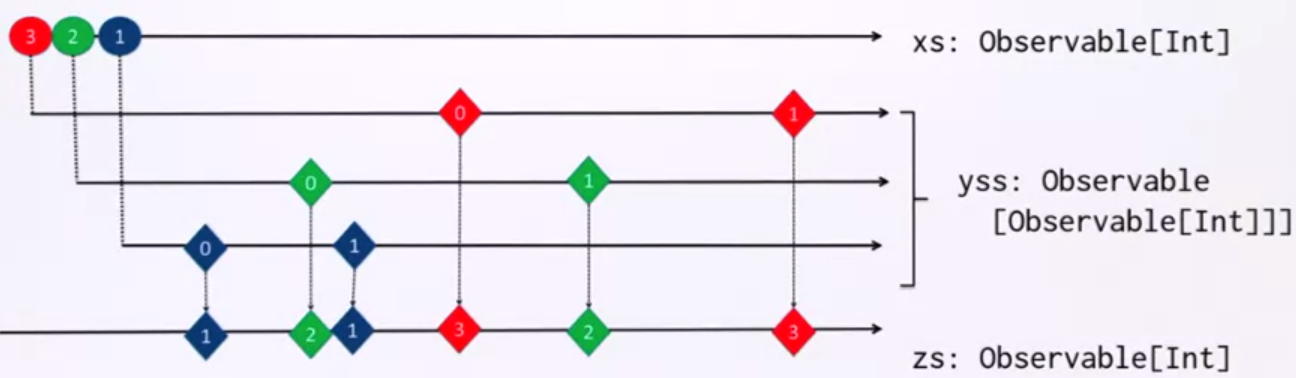
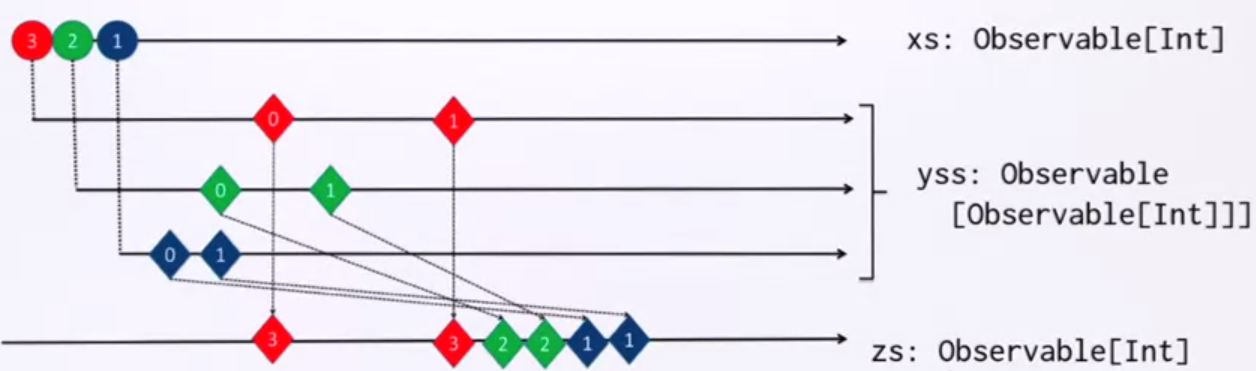
There is also an overloaded version of **window(long timespan, long timeshift, TimeUnit unit)** - it's creating a new window with the specific timespan each 'timeshift' period. So in this case there may be overlaping windows.

### Flatmap operator

Now the tricky part is that we need to register for events on each stream. This is where the **flatMap** operator comes in.   
The **flatMap** operator is like the swiss army knife, it can be used for all sorts of usecases.   
**1. The fork-join**. Consider the common case where for an event we need to make remote calls to other services. Each of those remote calls might return list of objects(but in the reactive streams world you might want to return Observable instead of List). So each event might start other streams and we register the downstream for notifications for each event.

**T --> Observable<T>**

This looks **similar to a fork-join** operation.

So the **3,2,1** events make remote calls that each return an Observable<Integer> while zs is the internal stream(the flatten operation) that downstreams operators subscribe on.   
We'll see later a concrete example of this(by making a rest call to mock REST services)   
Sidenote: The thing to notice from the image is that it has no longer an order guarantee. The events from the merged streams appear as soon as they are emitted and so you might get 'mixed' events between the streams.   
PS: Should we want to maintain order we can use **concatMap** like bellow:   


**2.The stream 'walker'** - There is already a stream of streams who emit events, we want the downstream operators to react on events that appears on each stream. This is the usecase we need right now. As a rule of thumb when you have an Observable<Observable<T>> you probably need flatMap next.

Observable<Observable<JsonObject>> windowStream = failedLogins().window(30, TimeUnit.SECONDS);

Observable<JsonObject> windowEvents = windowStream.flatMap(jsonWindowEventObservable -> jsonWindowEventObservable); //notice we now have a single stream as return value

The statement

flatMap(jsonWindowEventObservable -> jsonWindowEventObservable)

looks like it's not doing anything, but we just say that the map function should return our json event unchanged, but we now have the events from all the windows flattened into a single stream.

Since we want to count the number of failed logins per ip, we need the **groupBy** operator. It takes as parameter a function which gives the grouping key, in our case a function which returns the 'remoteIP' field in the json event for failed logins.   
The **groupBy** operator returns a stream of streams, each new ip starts a new stream, while already "seen" ips emit events on their stream.

Observable<GroupedObservable<String, JsonObject>> groupedEventsPerIp = windowEvents.groupBy(Json.propertyStringValueFunc("remoteIP"));

192.168.0.11 -(#)--------(#)----->   
192.168.0.68 ---------(#)-------->   
192.168.0.88 -----(#)----------->

This is another reason why we first chose to use **window** on the initial stream of log events.   
Since the initial stream of potentially infinite, groupBy splits it into potentially infinite number of streams aka a recipe for OutOfMemory.

Now that we have to count the number of events per each ip-substream. We again use flatMap to be able to register a function for each substream perform some calculation and then "flatten" the result back to a single stream.

.flatMap(groupEventForIP -> groupEventForIP

.count()

.map(failedLoginsCount -> {

final String remoteIp = grouped.getKey();

return new Pair<>(remoteIp, failedLoginsCount);

}))

The flatMap operator has received as a parameter a groupEventForIP which is a GroupedObservable<String, JsonObject> (the string key being the remoteIP value) - so it's still a stream which means we can go ahead and apply stream operators on this substream-.   
Since we need to count how many failed logins there were, we just apply the **count()** operator which returns a number that we use to pass downstream a Pair<remoteIP, failedLoginsCount>.

All together this looks like:

Observable<Pair<String, Integer>> failedLoginsPerWindow(int windowSecs) {

return failedLogins()

.window(windowSecs, TimeUnit.SECONDS)

.flatMap(jsonWindowEventObservable -> jsonWindowEventObservable

.groupBy(Json.propertyStringValueFunc("remoteIP"))

.flatMap(groupEventForIP -> groupEventForIP

.count()

.map(failedLoginsCount -> {

final String remoteIp = groupEventForIP.getKey();

return new Pair<>(remoteIp, failedLoginsCount);

}))

);

}

@Bean

public Subscription failedLoginsPerIpSubscription() {

return failedLoginsPerWindow(10) //10 secs window

.filter(failedLoginPair -> failedLoginPair.getValue() > minFailedLoginAttempts)

.subscribe((failedLoginPair) ->log.info("Possible brute force login: {}", failedLoginPair));

}

#### Usecase for flatMap as the mentioned fork-join operator to do extra checks

We dealt with the bad guys, however we've yet to seen how we can deal with a scenario in which we need to run extra checks in remote services for certain events.   
Let's invent some usecase.   
Say that after login it would make sense to start some credit check for our user like verify he payed at least one bill or that he registered a valid mobile phone with us. Since these checks are usually slow(involving some legacy core business components) and change frequently, we don't want to put them in our frontend app to slow down the load time of pages.   
Instead we can do these checks on our log stream processing machines and write the results in some "isAllowedToXXX" field.   
The frontend app would just check these fields before letting the user use services that have money costs. Since the checks logic is external, we can change it frequently without redeploying the frontend app.   
The user would be free to use his account to look around and just force him to enter extra data whenever he attempts to actually use those money involving services.

This would look like:

public class UserService {

public Observable<UserScoring> retrieveScoring(String username) {..}

}

Now we need another check to see how we can trigger them in parallel and still work with the result from both of them.   
Some sites when you travel to another country and see you login from an IP this country you've never logged from recently ask for some 2Factor authentication(like a sms code they send to your phone).   
This is a good idea since a login from a different country could mean someone else stole your credentials and is accessing your account.   
Now this is normally baked right in the authentication flow and not left to an external log event analysis tool, still it makes a good excuse to imagine it as our second extra check that the IP is from an user's country.

public class UserService {

/\*\* Returns UserCreditScoring.CREDIT\_CARD\_STORED, MOBILE\_PHONE\_STORED, NONE; \*/

public Observable<UserCreditScoring> retrieveScoring(String username) { }

/\*\* Returns UserLocationRating.SAFE, SUSPICIOUS \*/

public Observable<UserLocationRating> retrieveUserLocationBasedOnIp(String username, String ip) { }

}

So for every successful login we want to start these two checks and in the case both of them return something other than SAFE / CREDIT-CARD-STORED we mark the check for the frontend to ask extra info.

But first we need to modify

JsonObject(json successful login event) -> Pair<Username, IP>

We need a simple **map** operator. Compared to **flatMap**, the **map** operator just transforms T -> X.

Observable<Pair<String, String>> successLoginStream =

succesfullLogins()

.map(jsonObject ->

new Pair<>(new Json(jsonObject).propertyStringValue("userName"),

new Json(jsonObject).propertyStringValue("remoteIP")))

and we got the two checks:

Observable<UserCreditScoring> scoringObservable = userService.retrieveScoring(username);

Observable<UserLocationRating> locationObservable = userService.retrieveUserLocationBasedOnIp(username, ip);

We need to wait for the result of both scoring and location checks stream.   
You might remember the **zip** operator that groups together one by one events but since they are different types of events(UserCreditScoring and UserLocationRating) we need a way to merge them together.

So we'll create a method that returns a Tuple(from [JavasLang](http://www.javaslang.io/) library) of the 3 values **<username, UserCreditScoring, UserLocationRating>**

private Observable<Tuple3<String, UserCreditScoring, UserLocationRating>> checkUserScoringAndLocationRating(String username, String ip) {

Observable<UserCreditScoring> scoringObservable = userService.retrieveScoring(username);

Observable<UserLocationRating> locationObservable = userService.retrieveUserLocationBasedOnIp(username, ip);

Observable<Tuple3<String, UserCreditScoring, UserLocationRating>> scoringAndLocationObservable =

scoringObservable

.zipWith(locationObservable, (scoring, location) -> new Tuple3<>(username, scoring, location));

return scoringAndLocationObservable;

}

we passed to the **zip** operator a merging function to say how we want to combine the scoring and location. This function just creates a new Tuple

(scoring, location) -> new Tuple3<>(username, scoring, location)

Remember that for **T -> Observable<Y>** we need **flatMap** to do the fork-join logic(to spin off new streams and then rejoining the results)

succesfullLogins()

.map(jsonObject ->

new Pair<>(new Json(jsonObject).propertyStringValue("userName"),

new Json(jsonObject).propertyStringValue("remoteIP")))

.flatMap(pair -> checkUserScoringAndLocationRating(pair.getKey(), pair.getValue()))

### RxJava Schedulers

I'm gonna add some debugging statements and show the output to understand something else. Btw we also need to subscribe to the observable first

private Observable<Tuple3<String, UserCreditScoring, UserLocationRating>> userScoringAndLocationForSuccessfulLogins() {

return succesfullLogins()

.map(jsonObject ->

new Pair<>(new Json(jsonObject).propertyStringValue("userName"),

new Json(jsonObject).propertyStringValue("remoteIP")))

.doOnNext(userIpPair -> log.info("After map {}", userIpPair))

.flatMap(pair ->

checkUserScoringAndLocationRating(pair.getKey(), pair.getValue())

.doOnNext(tuple3 -> log.info("Tupple {}", tuple3)) //-->> the extra debug statement

);

}

[json-hardcoded-file1] INFO JsonFileEmitterSourceConfiguration - Emitting SUCCESS login for user='mendoza@yahoo.com'

[json-hardcoded-file1] INFO SuccessfulLoginObservable - Tupple (mendoza@yahoo.com, MOBILE\_PHONE\_STORED, SAFE)

[json-hardcoded-file1] INFO SuccessfulLoginObservable - Subscriber got (mendoza@yahoo.com, MOBILE\_PHONE\_STORED, SAFE) \*\*\*\*

[json-hardcoded-file1] INFO JsonFileEmitterSourceConfiguration - Emitting FAILED login for user='perez@yahoo.com'

[json-hardcoded-file1] INFO JsonFileEmitterSourceConfiguration - Emitting SUCCESS login for user='powell@yahoo.com'

[json-hardcoded-file1] INFO SuccessfulLoginObservable - Tupple (powell@yahoo.com, MOBILE\_PHONE\_STORED, SAFE)

[json-hardcoded-file1] INFO SuccessfulLoginObservable - Subscriber got (powell@yahoo.com, MOBILE\_PHONE\_STORED, SAFE) \*\*\*\*

[json-hardcoded-file1] INFO JsonFileEmitterSourceConfiguration - Emitting SUCCESS login for user='walton@gmail.com'

[json-hardcoded-file1] INFO SuccessfulLoginObservable - Tupple (walton@gmail.com, NONE, SAFE)

[json-hardcoded-file1] INFO SuccessfulLoginObservable - Subscriber got (walton@gmail.com, NONE, SAFE) \*\*\*\*

The thing to notice is that all the operations are happening on 'json-hardcoded-file1' which is the same thread on which we read the json events so if we'd be doing some slow processing with the events we'd keep all this thread blocked and slow down receiving of events.   
Can we maybe make part of the work to run in parallel?

RxJava provides some high level concepts for concurrent execution, like ExecutorService we're not dealing with the low level constructs like creating the Threads ourselves. Instead we're using **Schedulers** which create Workers who are responsible for scheduling and running code. By default RxJava will not introduce concurrency and will run the operations on the subscription thread.

There are two methods through which **we can introduce Schedulers** into our chain of operations: **subscribeOn** and **observeOn**.

* **subscribeOn** acts upon the creation of events part -> at Observable.create and it will use the changed thread for the downstream operations.

Observable<Integer> observable = Observable

.create(subscriber -> {

/\*\* some possible slow network op\*\*/

log.info("Started emitting");

subscriber.onNext(1);

subscriber.onCompleted();

});

observable

// .subscribeOn(Schedulers.computation()) //--> changes the thread where create above runs

.map(val -> {

int newValue = val \* 2;

log.info("Mapping new val {}", newValue);

return newValue;

})

.subscribe(val -> log.info("Subscribe received " + val));

Short recap: **Observable are lazy**(the code inside the first line it's not executed) until we call **observable.subscribe()** and then the thread which called **subscribe()** will be used to run the code inside **create** along with the slow network op unless ofcourse we specified another scheduler with **subscribeOn** to execute it.

with the subscribeOn line commented out we have:

[main] INFO SchedulerTests - Starting

[main] INFO SchedulerTests - Started emitting

[main] INFO SchedulerTests - Mapping new val 2

[main] INFO SchedulerTests - Subscribe received 2

while with subscribeOn enabled:

[main] INFO SchedulerTests - Starting

[RxComputationScheduler-1] INFO SchedulerTests - Started emitting

[RxComputationScheduler-1] INFO SchedulerTests - Mapping new val 2

[RxComputationScheduler-1] INFO SchedulerTests - Subscribe received 2

Since subscribeOn is related to which thread the **Observable.create** method is invoked on - and not with the code inside **.subscribe(val -> { ... })** part(which is the last operation in our chain), it doesn't actually matter where we call **subscribeOn** in our method chain, neither does it matter if we call it twice(just the first one will count). And very important in our case **calling subscribeOn for Subjects is pointless** since hot observables are already pushing events on some thread even without an active subscriber registered.

* **observeOn** relates to how subsequent operators are being executed.

Observable<Integer> observable = Observable

.create(subscriber -> {

log.info("Started emitting");

subscriber.onNext(1);

subscriber.onCompleted();

})

.observeOn(Schedulers.computation()) //--> change subsequent operations thread

.map(val -> {

int newValue = val \* 2;

log.info("First mapping new val {}", newValue);

return newValue;

})

.observeOn(Schedulers.io()) // --> change it again

.map(val -> {

String newValue = "\*" + val + "\*";

log.info("Second mapping new val {}", newValue);

return newValue;

})

.subscribe(val -> log.info("Subscribe received " + val));

we get

12:49:31 [main] INFO SchedulerTests - Started emitting

12:49:31 [RxComputationScheduler-1] INFO SchedulerTests - First mapping new val 2

12:49:31 [RxIoScheduler-2] INFO SchedulerTests - Second mapping new val \*2\*

12:49:31 [RxIoScheduler-2] INFO SchedulerTests - Subscribe received \*2\*

because the subscription happened on the 'main' thread, while before each operator we changed the Scheduler.

RxJava provides some **out of the box Schedulers**:

* **Schedulers.newThread()** - The scheduler just starts a new thread every time it is requested via subscribeOn() or observeOn(). Not usually a good choice since threads are not reused, and a new one is always created.
* **Schedulers.io()** - as the name implies it's supposed to be used for blocking network/disk code. Nothing really magical, it's a threadpool with an unbounded number of threads. Threads are being reused, but since this will be a common pool(just as using the same ForkJoinPool for all calls to Streams.parallel() is a bad idea), it's probably good to be using it sparsely.
* **Schedulers.computation()** - should be used on operations that require computational power and have no blocking code. It's limiting the pool size with the number of cores so there are no more concurrent threads running than CPU cores and all other requests are queued up until a thread becomes available.
* **Schedulers.from(Executor executor)** - provides a Scheduler based on an Executor.

So back to our code, we just need to mention a Scheduler upfront and the subsequent operations will use a worker from it.

@Bean

Observable<JsonObject> loginEvents() {

return events

.observeOn(Schedulers.io())

.filter(checkPropertyFunc("logger\_name", val -> val.contains("LoginEvent")));

}

Next we're going to see how we can call a REST service the reactive way to make good on our promise of less used threads for increased performance of many events handling.

## [Reactive Programming with Spring Reactor](https://ordina-jworks.github.io/reactive/2016/12/12/Reactive-Programming-Spring-Reactor.html)

Posted Dec 12, 2016 in [Reactive](https://ordina-jworks.github.io/categories/reactive/) by [Tom Van den Bulck](https://ordina-jworks.github.io/author/tom-van-den-bulck)   
[JOIN](https://ordina-jworks.github.io/tags/join/), [Spring](https://ordina-jworks.github.io/tags/spring/), [Reactor](https://ordina-jworks.github.io/tags/reactor/), [Java](https://ordina-jworks.github.io/tags/java/), [Reactive](https://ordina-jworks.github.io/tags/reactive/)

# Overview

* [Stephane Maldini @ JOIN](https://ordina-jworks.github.io/reactive/2016/12/12/Reactive-Programming-Spring-Reactor.html#join)
* [The new normal that is not new](https://ordina-jworks.github.io/reactive/2016/12/12/Reactive-Programming-Spring-Reactor.html#the-new-normal)
* [The Reactive Manifesto](https://ordina-jworks.github.io/reactive/2016/12/12/Reactive-Programming-Spring-Reactor.html#the-manifesto)
* [Latency & Blocking](https://ordina-jworks.github.io/reactive/2016/12/12/Reactive-Programming-Spring-Reactor.html#latency)
* [The Contract](https://ordina-jworks.github.io/reactive/2016/12/12/Reactive-Programming-Spring-Reactor.html#contract)
* [Reactive Types](https://ordina-jworks.github.io/reactive/2016/12/12/Reactive-Programming-Spring-Reactor.html#reactive-types)
* [Testing & Debuging](https://ordina-jworks.github.io/reactive/2016/12/12/Reactive-Programming-Spring-Reactor.html#testing)
* [Other Changes](https://ordina-jworks.github.io/reactive/2016/12/12/Reactive-Programming-Spring-Reactor.html#changes)
* [RxJava](https://ordina-jworks.github.io/reactive/2016/12/12/Reactive-Programming-Spring-Reactor.html#rxjava)
* [Spring Framework 5](https://ordina-jworks.github.io/reactive/2016/12/12/Reactive-Programming-Spring-Reactor.html#spring-5)
* [Conclusion & Do It Yourself](https://ordina-jworks.github.io/reactive/2016/12/12/Reactive-Programming-Spring-Reactor.html#conclusion)

# Stephane MaldiniStephane Maldini @ JOIN 2016

On 5 October 2016, we had the pleasure to welcome Stephane Maldini at our [JOIN event](https://ordina-jworks.github.io/conferences/2016/09/27/JOIN-2016.html).

A multi-tasker eating tech 24/7, Stephane is interested in cloud computing, data science and messaging. Leading the Reactor Project, Stephane Maldini is on a mission to help developers create reactive and efficient architectures on the JVM and beyond. He is also one of the main contributors for Reactive support in the upcoming Spring 5 framework, which can be seen as the new standard for reactive applications in the Java world.

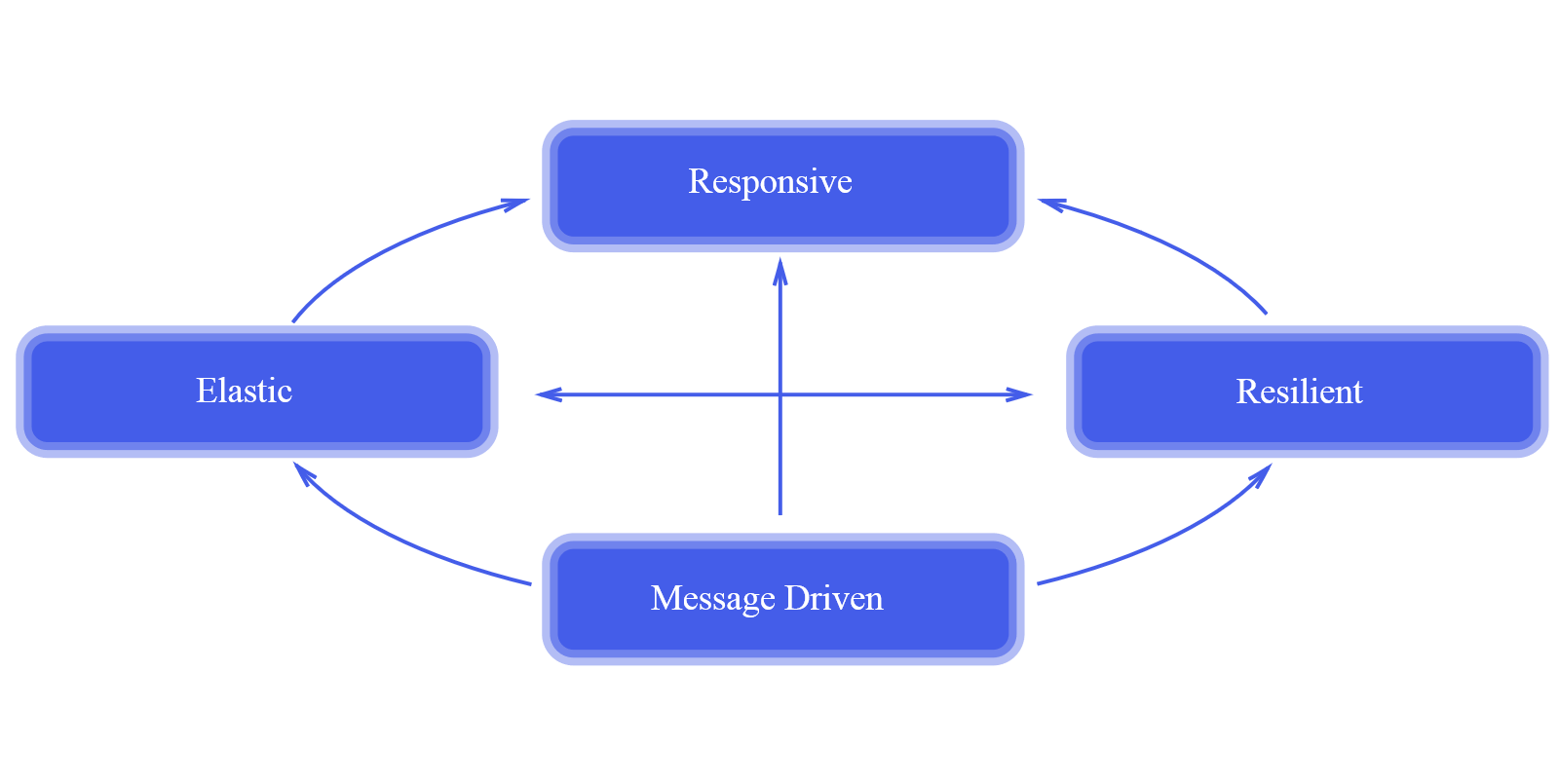
You can rewatch his [talk](https://www.youtube.com/watch?v=RU0yQhfybDg) on on our [Channel](https://www.youtube.com/channel/UCsebfWdqV7LqNNDMDvCESIA) on Youtube.

# The new normal that is not new

It has been around for 30-40 years and boils down to [Event-Driven Programming](https://en.wikipedia.org/wiki/Event-driven_programming) What is new is “reactive motion bound to specification”, this means that reactive programming is based on something solid, a specification and no longer some functional concepts. Namely the [Reactive Manifesto](http://www.reactivemanifesto.org/).

Because of this specification, Spring found it the right time to start with [Reactor](https://spring.io/blog/2013/05/13/reactor-a-foundation-for-asynchronous-applications-on-the-jvm) as they could now build something, which would be able to work and where it was clear what people could expect.

# The Reactive Manifesto



According to the manifesto, reactive systems are

* **Responsive**: respond in a timely manner if at all possible, responsiveness means that problems can be detected quickly and dealt with accordingly.
* **Resilient**: remain responsive in the event of failure, failures are contained with each component isolating components from each other.
* **Elastic**: stay responsive under varying workload, reactive systems can react to changes in the input rate by increasing or decreasing the resources allocated to services.
* **Message Driven**: rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency.

This boundary also provides the means to delegate failures as messages.

Systems built as reactive systems are thus more flexible, loosely-coupled and scalable. This makes them easier to develop and to allow changes. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster.

# Latency

Latency is also a real issue, the real physical distance of various components and services becomes more important with cloud based systems. This is also a very random number which is difficult to predict because it can depend on network congestion. With [Zipkin](https://github.com/openzipkin/zipkin), you can measure this latency.

The same latency can also exist within an application - between the different threads - although the impact will be less severe than between various components.

Something needs to be done when latency becomes too big of an issue, especially if the receiver can not process enough. Too much data will fill up the buffer and can result, with an unbounded queue, to the infamous [OutOfMemoryException()](https://docs.oracle.com/javase/8/docs/api/java/lang/OutOfMemoryError.html). While you won’t run out of memory with a circular buffer, you risk losing messages as the oldest ones get overwritten.

# Blocking

One way to prevent out of memory exceptions is to use blocking.

But this can be a real poison pill: when a queue is full, it will block a thread and as more and more queues get blocked your server will die a slow death.

Blocking is faster and has better performance, than reactive, but reactive will allow for more concurrency. Concurrency is important if you have a microservice based architecture, as there you typically need to be more careful and more exact when allocating resources between services.

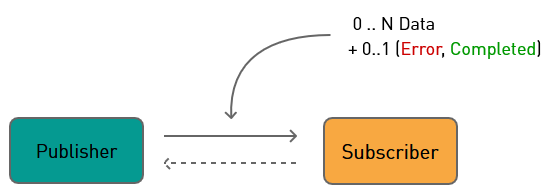
As in, by being more concurrent you can save a lot of money when using cloud and microservices.

# Contract

Reactive is non-blocking and messages will never overflow the queue, see for the standard definition <http://www.reactive-streams.org/>.

Created by Pivotal, Typesafe, Netflix, Oracle, Red Hat and others.

The scope of Reactive Streams is to find a minimal set of interfaces, methods and protocols that will describe the necessary operations and entities to achieve the goal—asynchronous streams of data with non-blocking [back-pressure](http://www.reactivemanifesto.org/glossary#Back-Pressure). With back-pressure, a consumer which can not handle the load of events sends towards it, can communicate this towards the upstream components so these can reduce the load. Without back-pressure the consumer would either fail catastrophically or drop events.



This contract defines to send data 0 .. N. [Publisher](http://www.reactive-streams.org/reactive-streams-1.0.0-javadoc/org/reactivestreams/Publisher.html) is an interface with a subscribe() method. [Subscriber](http://www.reactive-streams.org/reactive-streams-1.0.0-javadoc/org/reactivestreams/Subscriber.html) has 4 callback methods: onSubscribe(), onNext() (which can be called 0 to N times), onComplete() and onError(). The last two signals (complete and error) are terminal states, no further signals may occur and the subscriber’s subscription is considered cancelled.

What is important is the reverse flow and the back-pressure. After subscribing, the subscriber gets a subscription which is a kind of 1 on 1 relationship between the subscriber and the publisher with 2 methods: request and cancel.

* **Request**: this is the more important one, with this method the subscriber will ask the publisher to send x messages (and not more), a so called pull.
* **Cancel**: the subscription is being cancelled.

Spring Reactor focuses on the publisher side of the reactive streaming, as this is the hardest to implement and to get right. It provides you with the tools to implement publishers in a back-pressure way.

The publisher is a provider of a potentially unbounded number of sequenced elements, publishing them according to the demand received from its Subscriber(s).

The Reactive Streams specification has been adopted for [java 9](http://gee.cs.oswego.edu/dl/jsr166/dist/docs/java/util/concurrent/Flow.html).

# DIY Reactive Streams

Implementing a Reactive Stream framework yourself is very hard to do, for Stephane Maldini this is the 4th or 5th attempt. For Davik Karnok, the tech lead of RxJava, it is attempt 7 or 8. The main difficulty is to make it side effect free.

For example:

Publisher<User> rick = userRepository.findUser("rick");

Note that a publisher is returned instead of directly returning the entity. By doing so it does not block the subscribers when querying for the user and the publisher will produce the user when ready.

But by using the specification as is, your publisher might produce 0, 1 or N users, returning an Iterable as result. This is not really practical to work with, as most of the time we are only interested in a single user and not a stream of multiple results. When you would be building the method findOneUser() you also would not want to return an Iterable but just a single User.

Also you will have to implement a subscriber to define the action to perform when the result is available.

rick.subscribe(new Subscriber<User>(){...});

Implementing this subscriber would not be that hard, because the specification has been made so that all complexity lies at the publishers side.

Another issue is that you can only subscribe on the publisher, there are no other methods available like [map](http://martinfowler.com/articles/collection-pipeline/map.html), [flatmap](http://martinfowler.com/articles/collection-pipeline/flat-map.html), …

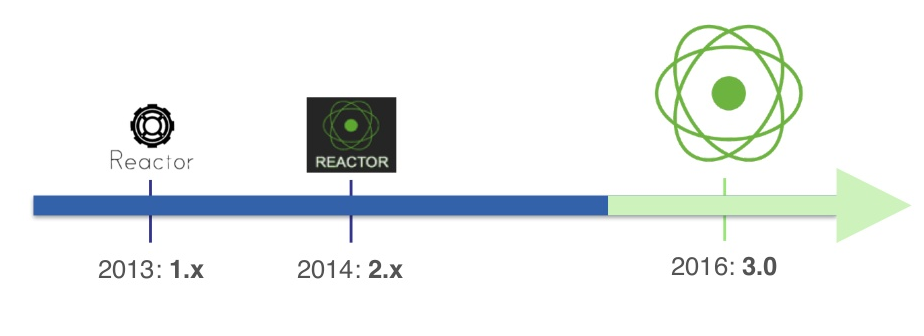
The other point is that when designing your own API you will also have to deal with the following issues:

* Should work with [RS TCK](https://jcp.org/en/jsr/detail?id=311) (otherwise it might not work with other libraries as well)
* Address reentrance
* Address thread safety
* Address efficiency
* Address state
* For Many-To-One flows, implement your own merging operation
* For One-To-Many flows, implement your own broadcasting operation
* …

This is all very hard to do yourself.

# 3 Years to Mature

It took Spring Reactor 3 years to mature.

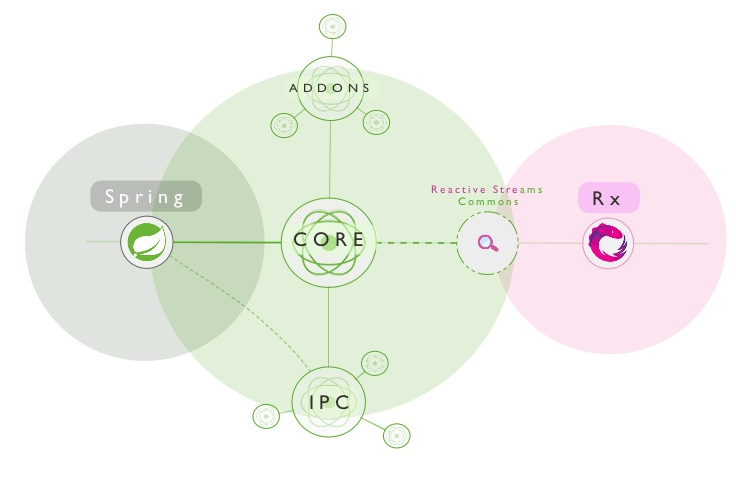


2.0 was not side effect free - also existential questions were raised around the project. At the same time Spring evolved and microservices became the norm.

Spring needs to work nicely with these microservices, concurrency is important, can Reactor not be used for that?

With 3.0 the team wanted to focus on microservices, take some ideas from [Netflix OSS](https://netflix.github.io/) and implement these in a pragmatic way. Actually Reactor 3 was started as 2.5, but so many new features were added that the version had to be changed as well in order to reflect this.

Since 3.0 Spring Reactor has been made more modular and consists of several components:



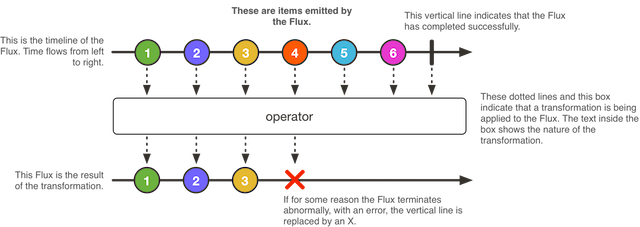
* [Core](https://github.com/reactor/reactor-core) is the main library. Providing a non-blocking Reactive Streams foundation for the JVM both implementing a [Reactive Extensions](https://github.com/Reactive-Extensions) inspired API and efficient message-passing support.
* [IPC](https://github.com/reactor/reactor-ipc): back-pressure-ready components to encode, decode, send (unicast, multicast or request/response) and serve connections. Here you will find support for [Kafka](https://kafka.apache.org) and [Netty](http://netty.io).
* [Addons](https://github.com/reactor/reactor-addons): Bridge to RxJava 1 or 2 Observable, Completable, Flowable, Single, Maybe, Scheduler, and also Swing/SWT Scheduler, Akka Scheduler.
* [Reactive Streams Commons](https://github.com/reactor/reactive-streams-commons) is the research project between Spring Reactor and RxJava as both teams had a lot of ideas they wanted to implement. Lots of effort was put in order to create real working, side-effect free operations. [Map](https://martinfowler.com/articles/collection-pipeline/flat-map.html) and [Filtering](http://martinfowler.com/articles/collection-pipeline/filter.html) for example are easy, but mergings, like [Flatmap](https://martinfowler.com/articles/collection-pipeline/flat-map.html) are hard to implement side-effect free. Having a proper implementation in the research project for these operations allowed the team to experiment and make it quite robust. This project contains Reactive-Streams compliant operators, which in turn are implemented by Spring Reactor and RxJava. Both the Spring and RxJava teams are very happy with this collaboration and this is still continuing. When a bug gets fixed in Spring Reactor it will also be fixed in RxJava and vice versa.

Everything in Reactor is just reactive streams implementation - which is used for the reactive [story](https://spring.io/blog/2016/07/28/reactive-programming-with-spring-5-0-m1) of spring 5.

There also exists an implementation for .NET, [Reactor Core .NET](https://github.com/reactor/reactor-core-dotnet) and one for javascript [Reactor Core TypeScript](https://github.com/reactor/reactor-core-js).

# Reactive Types

## Flux vs Observable



Observable is not implementing Reactive Streams Publisher which means that if you would like to use the Spring 5 save(Publisher<T>) you first have to convert the Observable to a Flowable as you can see in [Observable and Flowable](https://github.com/ReactiveX/RxJava/wiki/What%27s-different-in-2.0#observable-and-flowable).

This was too much noise for the Spring team, they are less dependant on Android developers so they could go all in with Java 8.

Flux is a Reactive Streams Publisher with basic flow operations, where you start from a static method which will describe how the data will be generated, [just()](http://next.projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#just-T...-) is the simplest way

After that you have other operators like [Flatmap()](http://next.projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#flatMap-java.util.function.Function-), [Map()](http://next.projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#map-java.util.function.Function-), … to work with that data Some of the method names will be different to RxJava2, but the logic behind these methods has been aligned among RxJava and Spring .

Flux.just("red", "white", "blue")

.flatMap(carRepository::findByColor)

.collect(Result:: new, Result::add)

.doOnNext(Result::stop)

.subscribe(doWithResult);

Interface CarRepository {

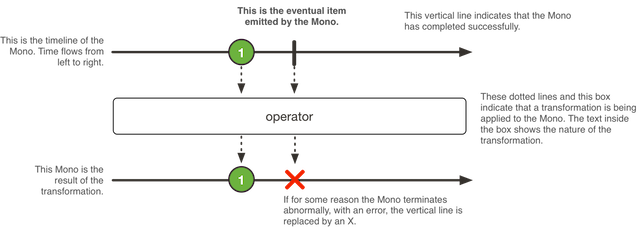
Flux<Car> findByColor(String color);

}

This Flux will retrieve all cars which match the color “red” then those with the color “white” and finally “blue”. So instead of just three elements, after this Flatmap we are going to have a lot more elements. This is all handled with back-pressure in mind, for example when the flatmap is busy merging data we will not ask for extra records

If the Repository implements Flux as a method signature, it will be picked up automatically as a reactive repository. This support for Flux will be part of the whole of Spring 5. Spring Data, Spring Security, Spring MVC, … are all good candidates who will have this kind of support.

## Mono



None is like a flux, but will return at most 1 result, so it does have less methods.

Mono.delayMillis(3000)

.map(d -> "Spring 4")

.or(Mono.delayMillis(2000).map(d -> "Spring 5"))

.then(t -> Mono.just(t + " world"))

.elapsed()

.subscribe()

This Mono will wait for 3 seconds on the “call” to Spring 4 or 2 seconds on that of Spring 5. The fastest result will be the one which will be outputted.

The Mono has as advantage over an Observable Future of Java 8 that a Mono will only be triggered if you subscribe to it. While with an Observable the call to send() will execute the operation.

# Testing

Block() exists for very specific use cases and for testing. Never, ever use this in production, as is it blocks your call, which does infer with the Reactive non-blocking statements. ;-)

Mono.delayMillis(3000)

.map(d -> "Spring 4")

.or(Mono.delayMillis(2000).map(d -> "Spring 5"))

.then(t -> Mono.just(t + " world"))

.elapsed()

.block()

You can also make use of [Stepverifier](ttp://next.projectreactor.io/ext/docs/api/reactor/test/StepVerifier.html) to test Flux, Mono and any other kind of Reactive Streams Publisher.

@Test

public void expectElementsWithThenComplete() {

expectSkylerJesseComplete(Flux.just(new User("swhite", null, null), new User("jpinkman", null, null)));

}

Use StepVerifier to check that the flux parameter emits a User with “swhite” username and another one with “jpinkman” then completes successfully.

void expectSkylerJesseComplete(Flux<User> flux) {

StepVerifier.create(flux)

.expectNextMatches(user -> user.getUsername().equals("swhite"))

.expectNextMatches(user -> user.getUsername().equals("jpinkman"))

.expectComplete();

}

# Debug

When you use reactive libraries you will quickly realize that step debugging is hard especially when you try to read your stacktraces, there are a lot of recursive calls taking place.

Before you invoke your operations you can enable an, expensive, debug mode.

Hooks.onOperator(op -> op.operatorStacktrace());

try {

Mono.just("a")

.map(d -> d)

.timestamp()

. ...

} catch (Exception e) {

e.printStacktrace()

}

When an exception is returned it will contain the exact operation that failed and the backtrace to that operation.

You must enable this [Hooks.onOperator](https://projectreactor.io/core/docs/api/reactor/core/publisher/Hooks.OperatorHook.html#operatorStacktrace--) before the operations you want to track.

# More cool stuff

## ParallelFlux

If you want to stress test your CPU you can use [ParallelFlux](http://projectreactor.io/core/docs/api/reactor/core/publisher/ParallelFlux.html) which will spread the workload in concurrent tasks when possible.

Mono.fromCallable( () -> System.currentTimeMillis() )

.repeat()

.parallel(8) //parallelism

.runOn(Schedulers.parallel())

.doOnNext( d -> System.out.println("I'm on thread "+Thread.currentThread()) ).

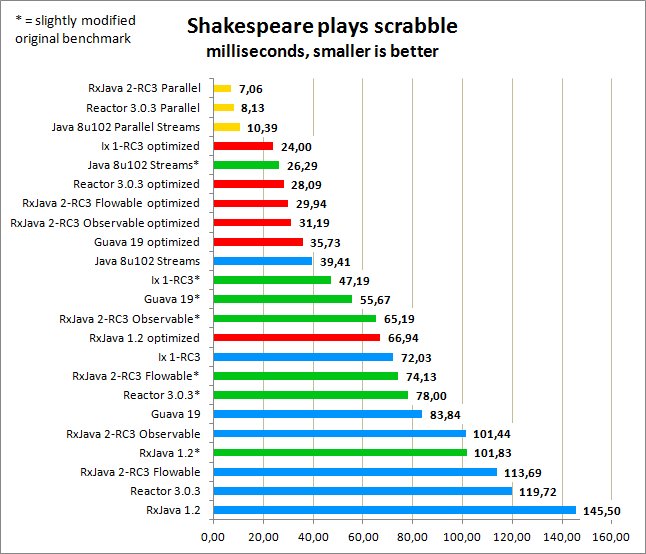
.sequential()

.subscribe()

This basically avoids that you have to write flatMap(), where after the parallel(x) you will have exactly x number of Rails or Flux. Afterwards you can merge these back into a Flux with sequential().

A nice feature is that it keeps the code more readable with everything on a single indentation level.

But the cool part is that it is also very performant, with parallel, Reactor is very close to the bare metal of what the JVM can do as you can see in the below comparisation:

  
<https://twitter.com/akarnokd/status/780135681897197568>

## Bridge Existing Async code

To bridge a Subscriber or Processor into an outside context that is taking care of producing non concurrently, use Flux.create(), Mono.create(), or FluxProcessor.connectSink().

Mono<String> response = Mono.create( sink -> {

HttpListener listener = event -> {

if (event.getResponseCode() >= 400) {

sink.error(new RunTimeException("Error"));

} else {

String result = event.getBody();

if (body.isEmpty()) {

sink.succes();

} else {

sink.success(body);

}

}

};

client.addListener(listener);

emitter.setCancellation(() -> client.removeListener(listener));

});

This create() allows you to bridge 1 result, which will be returned somewhere in the future, to a Mono.

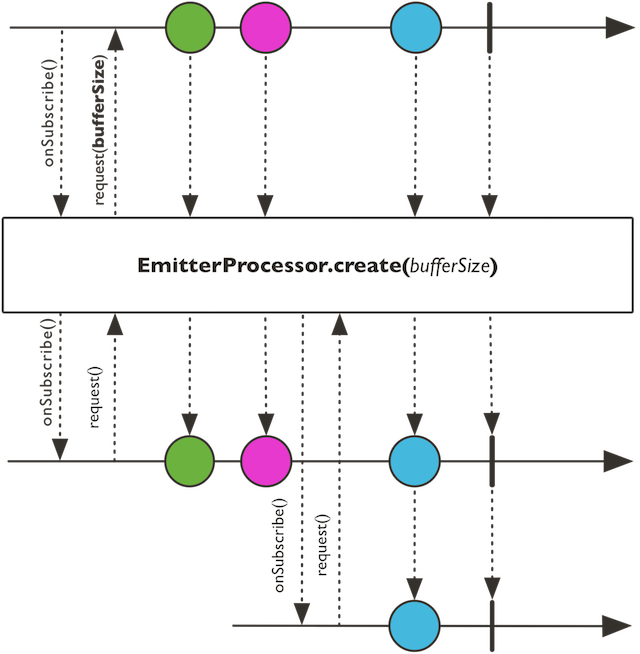
If you add a Kafka call, for example, where they have this callback so one can return onSuccess and onError you can use Mono.create(): see [Reactor Kafka](https://github.com/reactor/reactor-kafka) where this is used a lot.

Also exists for Flux of N items but it’s tougher and more dangerous as you must explicitly indicate what to do in the case of overflow; keep the latest and risk losing some data or keep everything with the risk of unbounded memory use. ¯\(ツ)/¯

## Create Gateways to Flux and Mono

There also exist some options to bridge the synchronous world with the Flux and the Mono.

Like for example the [EmitterProcessor](http://projectreactor.io/core/docs/api/?reactor/core/publisher/EmitterProcessor.html) which is a signal processor.



EmitterProcessor<Integer> emitter = EmitterProcessor.create();

BlockingSink<Integer> sink = emitter.connectSink();

sink.next(1);

sink.next(2);

emitter.subscribe(System.out::println);

sink.next(3); //output : 3

sink.finish();

But you also have:

* [ReplayProcessor](http://projectreactor.io/core/docs/api/?reactor/core/publisher/ReplayProcessor.html), a caching broadcaster.
* [TopicProcessor](http://projectreactor.io/core/docs/api/?reactor/core/publisher/TopicProcessor.html), an asynchronous signal broadcaster
* [WorkQueueProcessor](http://projectreactor.io/core/docs/api/?reactor/core/publisher/WorkQueueProcessor.html), which is similar to the TopicProcessor but distributes the input data signal to the next available Subscriber.

These are all an implementation of a [RingBuffer](https://en.wikipedia.org/wiki/Circular_buffer) backed message-passing Processor implementing publish-subscribe with synchronous drain loops.

## Optimizations

Operation fusion: Reactor has a mission to limit the overhead in stack and message passing. They distinguish 2 types of optimization:

* Macro Fusion: Merge operators in one during assembly time, for example, if the user does .merge() - .merge() - .merge() spring reactor is smart enough to put this in a single .merge()
* Micro Fusion: Because of the Reactive specification and the asynchronous nature of the response, queues are heavily used, but creating a queue for every request/response is very costly.

Spring Reactor will avoid to create queues whenever possible and short circuit during the lifecycle of the request. They are going to merge the queue from downstream with the one from upstream - hence the name fusion. If the parent is something we can pull (an Iterable or a queue) then Reactor is going to use the parent as a queue, thus avoiding to create a new queue. This is very smart to do - but also very complicated to do yourself, because Spring Reactor has this in place you do not have to deal with this hassle..

## A Simpler API

Reactor: a Simpler API, the entire framework just fits in 1 jar: reactor-core jar. Flux and Mono live in the [reactor.com](http://reactor.com).publisher package, reactor.core.scheduler contains the FIFO task executor.

By default the Publisher and Subscriber will use the same thread. With publishOn() the publisher can force the subscriber to use a different thread, while the subscriber can do the same with subscribeOn().

For Reactor 3.x there will be more focus on the [javadoc](http://projectreactor.io/core/docs/api), as this has been lagging behind compared to the new features which have been developed.

# RxJava

Why Reactor when there’s already [RxJava2](https://github.com/ReactiveX/RxJava/wiki/What%27s-different-in-2.0)?

RxJava2 is java 6 while for Reactor the Spring team decided to go all in and focus only on Java 8. This means that you can make use of all the new and fancy Java 8 features.

If you are going to use Spring 5, Reactor might be the better option.

But if you are happy with your RxJava2, there is no direct need to migrate to Reactor.

# Spring Framework 5

It will still be backwards compatible. You can just take your Spring 4 application, put Spring 5 behind it and you will be good to go.

But with Spring 5 you will be able to make use of the following new components/ [Spring Web Reactive](http://docs.spring.io/spring-framework/docs/5.0.0.M1/spring-framework-reference/html/web-reactive.html) and Reactive HTTP. Which under the hood support [Servlet 3.1](https://java.net/downloads/servlet-spec/Final/servlet-3_1-final.pdf), [Netty](http://netty.io/) and [Undertow](http://undertow.io/).

The annotations are still very similar but you just return a Mono, so the User can now be retrieved in a non-blocking way.

@GetMapping("/users/{login}")

public Mono<User> getUser(@PathVariable String login) {

return this.repository.getUser(login);

}

# Conclusion

Spring Reactor is a very interesting framework, after 3 iterations it has matured and gives you a good base to get started with Reactive Streams. With the upcoming support in Spring 5 it will also start to become more mainstream.

Therefore I can see no better way then to get your hands dirty and learn more about Spring Reactor yourself.

* [reactive-programming-part-I](https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape): Provides you with a clear description of what reactive programming is about and its use cases. But also the different ways about how people have implemented reactive programming (actor model, futures, … ) and more specifially the different frameworks which implement reactive programming in java.

Frameworks like: [Spring Reactor](https://projectreactor.io/), [Spring Framework 5](http://projects.spring.io/spring-framework/), [RxJava](https://github.com/ReactiveX/RxJava/wiki) , [Akka](http://akka.io/), [Reactive Streams](http://www.reactive-streams.org/) and [Ratpack](https://ratpack.io/).

* [reactive-programming-part-II](https://spring.io/blog/2016/06/13/notes-on-reactive-programming-part-ii-writing-some-code): You will learn the API by writing some code, how to control the flow of data and its processing.
* [reactive-programming-part-III](https://spring.io/blog/2016/07/20/notes-on-reactive-programming-part-iii-a-simple-http-server-application): Here you will focus on more concrete use case and write something useful, but also on some low level features which you should learn to treat with respect.
* [reactor-api-hands-on](https://github.com/reactor/lite-rx-api-hands-on): This hands-on will help you learn easily the lite Rx API provider by Spring Reactor. You just have to make the unit tests green.
* On [spring.io](https://spring.io) you can find more interesting blog posts which will give you more background around Spring Reactor and provide you with the resources to start coding.

# Chapter 4. Applying Reactive Programming to Existing Applications

Tomasz Nurkiewicz

Introducing a new library, technology, or paradigm to an application, be it greenfield or legacy codebase, must be a careful decision. RxJava is not an exception. In this chapter, we review some patterns and architectures found in ordinary Java applications and see how Rx can help. This process is not straightforward and requires a significant mindset shift, therefore we will carefully transform from imperative to functional and reactive style. Many libraries in Java projects these days simply add bloat without giving anything in return. However, you will see how RxJava not only simplifies traditional projects, but what kinds of benefits it brings to legacy platforms.

I am pretty sure that you’re already very excited about RxJava. Built-in operators and simplicity makes Rx an amazingly powerful tool for transforming streams of events. However, if you go back to your office tomorrow, you will realize that there are no streams, no real-time events from stock exchange. You can hardly find any events in your applications; it’s just a mash-up of web requests, databases, and external APIs. You are so eager to try this new RxJava-thing somewhere beyond Hello world. Yet it seems that there are simply no use cases in real life that justify using Rx. Yet, RxJava can be a significant step forward in terms of architectural consistency and robustness. You do not need to commit to reactive style top-to-bottom—this is too risky and requires too much work in the beginning. But Rx can be introduced at any layer, without breaking an application as a whole.

We take you through some common application patterns and ways by which you can enhance them with RxJava in noninvasive way, with the focus being on database querying, caching, error handling, and periodic tasks. The more RxJava you add in various places of your stack the more consistent your architecture will become.

# From Collections to Observables

Unless your platform was built recently in JVM frameworks like [Play](https://www.playframework.com), [Akka actors](http://akka.io), or maybe [Vert.x](http://vertx.io), you are probably on a stack with a servlet container on one hand, and JDBC or web services on the other. Between them, there is a varying number of layers implementing business logic, which we will not refactor all at once; instead, let’s begin with a simple example. The following class represents a trivial repository abstracting us from a database:

class PersonDao {

List<Person> listPeople() {

return query("SELECT \* FROM PEOPLE");

}

private List<Person> query(String sql) {

//...

}

}

Implementation details aside, how is this related to Rx? So far we have been talking about asynchronous events pushed from upstream systems or, at best, when someone subscribes. How is this mundane Dao relevant here? Observable is not only a pipe pushing events downstream. You can treat Observable<T> as a data structure, dual to Iterable<T>. They both hold items of type T, but providing a radically different interface. So, it shouldn’t come as a surprise that you can simply replace one with the other:

Observable<Person> listPeople() {

final List<Person> people = query("SELECT \* FROM PEOPLE");

return Observable.from(people);

}

At this point, we made a breaking change to the existing API. Depending on how big your system is, such incompatibility might be a major concern. Thus, it is important to bring RxJava into your API as soon as possible. Obviously, we are working with an existing application so that can’t be the case.

# BlockingObservable: Exiting the Reactive World

If you are combining RxJava with existing, blocking and imperative code, might need have to translate Observable to a plain collection. This transformation is rather unpleasant, it requires blocking on an Observable waiting for its completion. Until Observable completes, we are not capable of creating a collection. BlockingObservable is a special type that makes it easier to work with Observable in nonreactive environment. BlockingObservable should be your last choice when working with RxJava, but it is inevitable when combining blocking and nonblocking code.

In [Chapter 3](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch03.html#functional-operators-and-transformations), we refactored the listPeople() method so that it returns Observable<People> rather than List. Observable is not an Iterable in any sense, so our code no longer compiles. We want to take baby steps rather than massive refactoring, so let’s keep the scope of changes as minimal as possible. The client code could look like this:

List<Person> people = pesonDao.listPeople();

String json = marshal(people);

We can imagine the marshal() method pulling data from the people collection and serializing them to JSON. That’s no longer the case, we can’t simply pull items from Observable when we want. Observable is in charge of producing (pushing) items and notifying subscribers if any. This radical change can be easily circumvented with BlockingObservable. This handy class is entirely independent from Observable and can be obtained via the Observable.toBlocking() method. The blocking variant of Observable has superficially similar methods like single() or subscribe(). However, BlockingObservable is much more convenient in blocking environments that are inherently unprepared for the asynchronous nature of Observable. Operators on BlockingObservable typically block (wait) until the underlying Observable is completed. This strongly contradicts the main concept of Observables that everything is likely asynchronous, lazy, and processed on the fly. For example, Observable.forEach() will asynchronously receive events from Observable as they come in, whereas BlockingObservable.forEach() will block until all events are processed and stream is completed. Also exceptions are no longer propagated as values (events) but instead are rethrown in the calling thread.

In our case, we want to transform Observable<Person> back into List<Person> to limit the scope of refactoring:

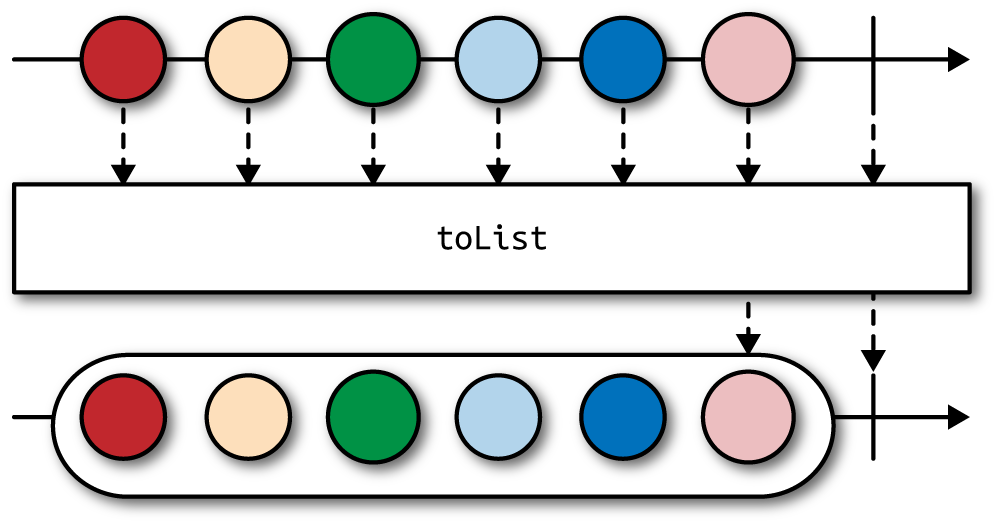
Observable<Person> peopleStream = personDao.listPeople();

Observable<List<Person>> peopleList = peopleStream.toList();

BlockingObservable<List<Person>> peopleBlocking = peopleList.toBlocking();

List<Person> people = peopleBlocking.single();

I intentionally left all intermediate types explicit in order to explain what happens. After refactoring to Rx, our API returns Observable<Person> peopleStream. This stream can potentially be fully reactive, asynchronous, and event driven, which doesn’t match at all what we need: a static List. As the first step, we turn Observable<Person> into Observable<List<Person>>. This lazy operator will buffer all Person events and keep them in memory until the onCompleted() event is received. At this point, a single event of type List<Person> will be emitted, containing all seen events at once, as illustrated in the following marble diagram:



The resulting stream completes immediately after emitting a single List item. Again, this operator is asynchronous; it doesn’t wait for all events to arrive but instead lazily buffers all values. The awkward looking Observable<List<Person>> peopleList is then converted to BlockingObservable<List<Person>> peopleBlocking. BlockingObservable is a good idea only when you must provide a blocking, static view of your otherwise asynchronous Observable. Whereas Observable.from(List<T>) converts normal pull-based collection into Observable, toBlocking() does something quite the opposite. You might ask yourself why we need two abstractions for blocking and nonblocking operators. The authors of RxJava figured out that being explicit about synchronous versus asynchronous nature of underlying operator is too crucial to be left for JavaDoc. Having two unrelated types ensures that you always work with the appropriate data structure. Moreover, BlockingObservable is your weapon of last resort; normally, you should compose and chain plain Observables as long as possible. However, for the purpose of this exercise, let’s escape from Observable right away. The last operator single() drops observables altogether and extracts one, and only one, item we expect to receive from BlockingObservable<T>. A similar operator, first(), will return a value of T and discard whatever it has left. single(), on the other hand, makes sure there are no more pending events in underlying Observable before terminating. This means single() will block waiting for onCompleted() callback. Here is the same code snippet as earlier, this time with all operators chained:

List<Person> people = personDao

.listPeople()

.toList()

.toBlocking()

.single();

You might think that we went through all this hassle of wrapping and unwrapping Observable for no apparent reason. Remember, this was just the first step. The next transformation will introduce some laziness. Our code as it stands right now always executes query("...") and wraps it with Observable. As you know by now, Observables (especially cold ones) are lazy by definition. As long as no one subscribes, they just represent a stream that never had a chance to begin emitting values. Most of the time you can call methods returning Observable and as long as you don’t subscribe, no work will be done. Observable is like a Future because it promises a value to appear in the future. But as long as you don’t request it, a cold Observable will not even begin emitting. From that perspective, Observable is more similar to java.util.function.Supplier<T>, generating values of type T on demand. Hot Observables are different because they emit values whether you are listening or not, but we are not considering them right now. The mere existence of Observable does not indicate a background job or any side effect, unlike Future, which almost always suggests some operation running concurrently.

# Embracing Laziness

So how do we make our Observable lazy? The simples technique is to wrap an eager Observable with defer():

public Observable<Person> listPeople() {

return Observable.defer(() ->

Observable.from(query("SELECT \* FROM PEOPLE")));

}

Observable.defer() takes a lambda expression (a factory) that can produce Observable. The underlying Observable is eager, so we want to postpone its creation. defer() will wait until the last possible moment to actually create Observable; that is, until someone actually subscribes to it. This has some interesting implications. Because Observable is lazy, calling listPeople() has no side effects and almost no performance footprint. No database is queried yet. You can treat Observable<Person> as a promise but without any background processing happening yet. Notice that there is no asynchronous behavior at the moment, just lazy evaluation. This is similar to how values in the [Haskell programming language](https://www.haskell.org/) are evaluated lazily only when absolutely needed.

If you never programmed in functional languages, you might be quite confused why laziness is so important and groundbreaking. It turns out that such behavior is quite useful and can improve the quality and freedom of your implementation quite a bit. For example, you no longer have to pay attention to which resources are fetched, when, and in what order. RxJava will load them only when they are absolutely needed.

As an example take this trivial fallback mechanism that we have all seen so many times:

void bestBookFor(Person person) {

Book book;

try {

book = recommend(person);

} catch (Exception e) {

book = bestSeller();

}

display(book.getTitle());

}

void display(String title) {

//...

}

You probably think there is nothing wrong with such a construct. In this example, we try to recommend the best book for a given person, but in case of failures, we degrade gracefully and display the best seller. The assumption is that fetching a bestseller is faster and can be cached. But what if you could add error handling declaratively so that try-catch blocks aren’t obscuring real logic?

void bestBookFor(Person person) {

Observable<Book> recommended = recommend(person);

Observable<Book> bestSeller = bestSeller();

Observable<Book> book = recommended.onErrorResumeNext(bestSeller);

Observable<String> title = book.map(Book::getTitle);

title.subscribe(this::display);

}

We are only exploring RxJava so far, thus I left all these intermediate values and types. In real life, bestBookFor() would look more like this:

void bestBookFor(Person person) {

recommend(person)

.onErrorResumeNext(bestSeller())

.map(Book::getTitle)

.subscribe(this::display);

}

This code is beautifully concise and readable. First find a recommendation for person. In case of error (onErrorResumeNext), proceed with a bestseller. No matter which one succeeded, map returns a value by extracting the title and then displays it. onErrorResumeNext() is a powerful operator that intercepts exceptions happening upstream, swallows them, and subscribes to provided backup Observable. This is how Rx implements a try-catch clause. We will spend much more time on error handling later in this book (see [“Declarative try-catch Replacement”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch07.html#Declarative-try-catch-replacement)). For the time being, notice how we can lazily call bestSeller() without worrying that fetching best seller happens even when a real recommendation went fine.

# Composing Observables

SELECT \* FROM PEOPLE is not really a state-of-the-art SQL query. First, you should not fetch all columns blindly, but fetching all rows is even more damaging. Our old API is not capable of paging results, viewing just a subset of a table. It might look like this, again in traditional enterprise application:

List<Person> listPeople(int page) {

return query(

"SELECT \* FROM PEOPLE ORDER BY id LIMIT ? OFFSET ?",

PAGE\_SIZE,

page \* PAGE\_SIZE

);

}

This is not a SQL book, so we’re going to set the implementation details aside. The author of this API was merciless: we don’t have the freedom to choose any range of records, we can only operate on 0-based page numbers. However in RxJava, due to laziness we can actually simulate reading an entire database starting from given page:

import static rx.Observable.defer;

import static rx.Observable.from;

Observable<Person> allPeople(int initialPage) {

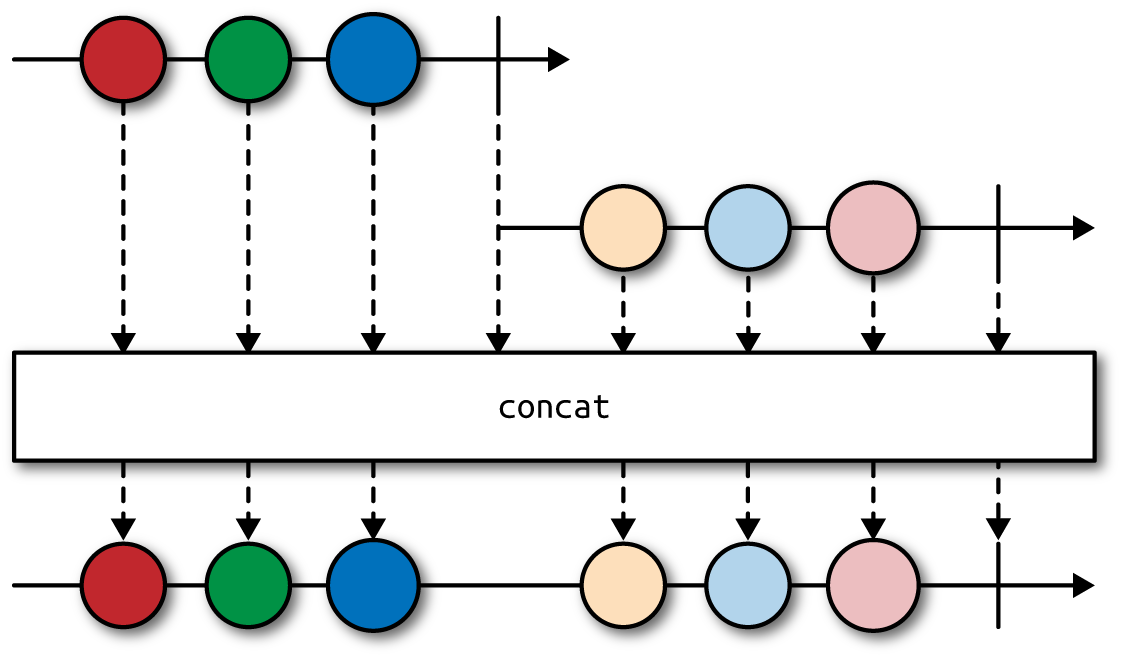
return defer(() -> from(listPeople(initialPage)))

.concatWith(defer(() ->

allPeople(initialPage + 1)));

}

This code snippet lazily loads the initial page of database records, for example 10 items. If no one subscribes, even this first query is not invoked. If there is a subscriber that only consumes a few initial elements (e.g., allPeople(0).take(3)), RxJava will unsubscribe automatically from our stream and no more queries are executed. So what happens when we request, say, 11 items but the first listPeople() call returned only 10? Well, RxJava figures out that the initial Observable is exhausted but the consumer is still hungry. Luckily, it sees concatWith() operator, that basically says: when the Observable on the left is completed, rather than propagating completion notification to subscribers, subscribe to Observable on the right and continue as if nothing happened, as depicted in the following marble diagram:



In other words, concatWith() can join together two Observables so that when the first one completes, the second one takes over. In a.concatWith(b).subscribe(...), subscriber will first receive all events from a, followed by all events from b. In this case, the subscriber first receives an initial 10 items followed by a subsequent 10. However, look carefully, there is an alleged infinite recursion in our code! allPeople(initialPage) calls allPeople(initialPage + 1) without any stop condition. This is a recipe for StackOverflowError in most languages, but not here. Again, calling allPeople() is always lazy, therefore the moment you stop listening (unsubscribe), this recursion is over. Technically concatWith() can still produce StackOverflowError here. Wait until [“Honoring the Requested Amount of Data”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch06.html#Honoring-requested-amount-of-data), you will learn how to deal with the varying demand for incoming data.

The technique of lazily loading data chunk by chunk is quite useful because it allows you to concentrate on business logic, not on low-level plumbing. We already see some benefits of applying RxJava even on a small scale. Designing an API with Rx in mind doesn’t influence the entire architecture, because we can always fall back to BlockingObservable and Java collections. But it’s better to have wide range of possibilities that we can further trim down if necessary.

## Lazy paging and concatenation

There are more ways to implement lazy paging with RxJava. If you think about it, the simplest way of loading paged data is to load everything and then take whatever we need. It sounds silly, but thanks to laziness it is feasible. First we generate all possible page numbers and then we request loading each and every page individually:

Observable<List<Person>> allPages = Observable

.range(0, Integer.MAX\_VALUE)

.map(this::listPeople)

.takeWhile(list -> !list.isEmpty());

If this were not RxJava, the preceding code would take an enormous amount of time and memory, basically loading the entire database to memory. But because Observable is lazy, no query to the database appeared yet. Moreover, if we find an empty page it means all further pages are empty, as well (we reached the end of the table). Therefore, we use takeWhile() rather than filter(). To flatten allPages to Observable<Person> we can use concatMap() (see [“Preserving Order Using concatMap()”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch03.html#Preserving-order-with-concatMap)):

Observable<Person> people = allPages.concatMap(Observable::from);

concatMap() requires a transformation from List<Person> to Observable<Person>, executed for each page. Alternatively we can try concatMapIterable(), which does the same thing, but the transformation should return an Iterable<Person> for each upstream value (happening to be Iterable<Person> already):

Observable<Person> people = allPages.concatMapIterable(page -> page);

No matter which approach you choose, all transformations on Person object are lazy. As long as you limit the number of records you want to process (for example with people.take(15)), the Observable<Person> will invoke listPeople() as late as possible.

# Imperative Concurrency

I don’t often see explicit concurrency in enterprise applications. Most of the time a single request is handled by a single thread. The same thread does the following:

* Accepts TCP/IP connection
* Parses HTTP request
* Calls a controller or servlet
* Blocks on database call
* Processes results
* Encodes response (e.g., in JSON)
* Pushes raw bytes back to the client

This layered model affects user latency when the backend makes several independent requests for instance to database. They are performed sequentially, whereas one could easily parallelize them. Moreover scalability is affected. For example in Tomcat there are 200 threads by default in the executors that are responsible for handling requests. This means that we can’t handle more than 200 concurrent connections. In case of a sudden but short burst of traffic, incoming connections are queued and the server responds with higher latency. However, this situation can’t last forever, and Tomcat will eventually begin rejecting incoming traffic. We will devote large parts of the next chapter (see [“Nonblocking HTTP Server with Netty and RxNetty”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch05.html#Non-blocking-HTTP-server-with-Netty)) on how to deal with this rather embarrassing shortcoming. For the time being, let’s stay with traditional architecture. Executing every step of request handling within a single thread has some benefits, for example improved cache locality and minimal synchronization overhead.[1](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#idm140239472529008) Unfortunately, in classic applications, because overall latency is the sum of each layer’s latencies, one malfunctioning component can have a negative impact on total latency.[2](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#idm140239472528016) Moreover, sometimes there are many steps that are independent from one another and can be executed concurrently. For example, we invoke multiple external APIs or execute several independent SQL queries.

JDK has quite good support for concurrency, especially since Java 5 with ExecutorService and Java 8 with CompletableFuture . Nonetheless, it is not as widely used as it could be. For example, let’s look at the following program with no concurrency whatsoever:

Flight lookupFlight(String flightNo) {

//...

}

Passenger findPassenger(long id) {

//...

}

Ticket bookTicket(Flight flight, Passenger passenger) {

//...

}

SmtpResponse sendEmail(Ticket ticket) {

//...

}

And on the client side:

Flight flight = lookupFlight("LOT 783");

Passenger passenger = findPassenger(42);

Ticket ticket = bookTicket(flight, passenger);

sendEmail(ticket);

Again, quite typical, classic blocking code, similar to what you can find in many applications. But if you look carefully from a latency perspective, the preceding code snippet has four steps; however, the first two are independent from each other. Only the third step (bookTicket()) needs results from lookupFlight() and findPassenger(). There exists an obvious opportunity to take advantage of concurrency. Yet, very few developers will actually go down this path because it requires awkward thread pools, Futures, and callbacks. What if the API were already Rx-compatible, though? Remember, you can simply wrap blocking, legacy code in Observable, just like we did in the beginning of this chapter:

Observable<Flight> rxLookupFlight(String flightNo) {

return Observable.defer(() ->

Observable.just(lookupFlight(flightNo)));

}

Observable<Passenger> rxFindPassenger(long id) {

return Observable.defer(() ->

Observable.just(findPassenger(id)));

}

Semantically, the rx- methods do exactly the same thing and in the same way; that is, they are blocking by default. We didn’t gain anything yet, apart from a more verbose API from the client perspective:

Observable<Flight> flight = rxLookupFlight("LOT 783");

Observable<Passenger> passenger = rxFindPassenger(42);

Observable<Ticket> ticket =

flight.zipWith(passenger, (f, p) -> bookTicket(f, p));

ticket.subscribe(this::sendEmail);

Both traditional blocking programs and the one with Observable work exactly the same way. It’s lazier by default, but the order of operations is essentially the same. First, we create Observable<Flight>, which as you already know, does nothing by default. Unless someone explicitly asks for a Flight, this Observable is just a lazy placeholder. We already learned that this is a valuable property of cold Observables. The same story goes for Observable<Passenger>; we have two placeholders of type Flight and Passenger, however no side-effects were performed yet. No database query or web-service call. If we decide to stop processing here, no superfluous work was done.

To proceed with bookTicket(), we need concrete instances of both Flight and Passenger. It is tempting to just block on these two Observables by using the toBlocking() operator. However, we would like to avoid blocking as much as possible to reduce resource consumption (especially memory) and allow greater concurrency. Another poor solution is to .subscribe() on the flight and passenger Observables and somehow wait for both callbacks to finish. It’s fairly straightforward when Observable is blocking, but if callbacks appear asynchronously and you need to synchronize some global state waiting for both of them, this quickly becomes a nightmare. Also a nested subscribe() is nonidiomatic, and typically you want a single subscription for one message flow (use case). The only reason why callbacks work somewhat decently in JavaScript is because there is just one thread. The idiomatic way of subscribing to multiple Observables at the same time is zip and zipWith. You might perceive zip as a way to join two independent streams of data pair-wise. But far more often, zip is simply used to join together two single-item Observables. ob1.zip(ob2).subscribe(...) essentially means that receiving an event when both ob1 and ob2 are done (emit an event on their own). So whenever you see zip, it’s more likely that someone is simply making a join step on two or more Observables that had forked paths of execution. zip is a way to asynchronously wait for two or more values, no matter which one appears last.

So let’s get back to flight.zipWith(passenger, this::bookTicket) (a shorter syntax using method reference instead of explicit lambda, as in the code sample). The reason I keep all of the type information rather than fluently joining expressions is because I want you to pay attention to return types. flight.zipWith(passenger, ...) doesn’t simply invoke callback when both flight and passenger are done; it returns a new Observable which you should immediately recognize as a lazy placeholder for data. Amazingly, at this point in time no computation was yet started, as well. We simply wrapped few data structures together, but no behavior was triggered. As long as no one subscribes to Observable<Ticket>, RxJava won’t run any backend code. This is what finally happens in last statement: ticket.subscribe() explicitly asks for Ticket.

# Where to Subscribe?

Pay attention to where you see subscribe() in domain code. Often your business logic is just composing Observables all the way down and returning them to some sort of framework or scaffolding layer. The actual subscription happens behind the scenes in a web framework or some glue code. It is not a bad practice to call subscribe() yourself, but try to push it out as far as possible.

To understand the flow of execution, it’s useful to look bottom up. We subscribed to ticket, thus RxJava must subscribe transparently to both flight and passenger. At this point the real logic happens. Because both Observables are cold and no concurrency is yet involved, the first subscription to flight invokes the lookupFlight() blocking method right in the calling thread. When lookupFlight() is done, RxJava can subscribe to passenger. However, it already received a Flight instance from synchronous flight. rxFindPassenger() calls findPassenger() in a blocking fashion and receives a Passenger instance. At this juncture, data flows back downstream. Instances of Flight and Passenger are combined using the provided lambda (bookTicket) and passed to ticket.subscribe().

This sounds like a lot of work considering it behaves and works essentially just like our blocking code in the beginning. But now we can declaratively apply concurrency without changing any logic. If our business methods returned Future<Flight> (or CompletableFuture<Flight>, it doesn’t really matter), two decisions would have been made for us:

* The underlying invocation of lookupFlight() already began and there is no place for laziness. We don’t block on such method, but work already started.
* We have no control over concurrency whatsoever, it is the method implementation that decides whether a Future task is invoked in a thread pool, new thread per request, and so on.

RxJava gives users more control. Just because Observable<Flight> wasn’t implemented with concurrency in mind, this does not mean that we cannot apply it later. Real-world Observables are typically asynchronous already, but in rare cases you must add concurrency to an existing Observable. The consumers of our API, not the implementors, are free to choose the threading mechanism in case of the synchronous Observable. All of this is achieved by using the subscribeOn() operator:

Observable<Flight> flight =

rxLookupFlight("LOT 783").subscribeOn(Schedulers.io());

Observable<Passenger> passenger =

rxFindPassenger(42).subscribeOn(Schedulers.io());

At any point before subscribing, we can inject subscribeOn() operator and provide a so-called Scheduler instance. In this case, I used the Schedulers.io() factory method, but we can just as well use a custom ExecutorService and quickly wrap it with Scheduler. When subscription occurs, the lambda expression passed to Observable.create() is executed within the supplied Scheduler rather than the client thread. It is not necessary yet but we will examine schedulers in depth in [“What Is a Scheduler?”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#What-is-a-Scheduler) section. For the time being, treat a Scheduler like a thread pool.

How does Scheduler change the runtime behavior of our program? Remember that the zip() operator subscribes to two or more Observables and waits for pairs (or tuples). When subscription occurs asynchronously, all upstream Observables can call their underlying blocking code concurrently. If you now run your program, lookupFlight() and findPassenger() will begin execution immediately and concurrently when ticket.subscribe() is invoked. Then, bookTicket() will be applied as soon as the slower of the aforementioned Observables emits a value.

Talking about slowness, you can declaratively apply a timeout, as well, when a given Observable does not emit any value in the specified amount of time:

rxLookupFlight("LOT 783")

.subscribeOn(Schedulers.io())

.timeout(100, TimeUnit.MILLISECONDS)

As always, in case of errors, they are propagated downstream rather than thrown arbitrarily. So if the lookupFlight() method takes more than 100 milliseconds, you will end up with TimeoutException rather than an emitted value sent downstream to every subscriber. The timeout() operator is exhaustively explained in [“Timing Out When Events Do Not Occur”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch07.html#Timing-out-when-events-do-not-occur).

We ended up with two methods running concurrently without much effort, assuming that your API is already Rx-driven. But we cheated a little bit with bookTicket() still returning Ticket, which definitely means it is blocking. Even if booking ticket was extremely fast, it is still worth declaring it as such, just to make the API easier to evolve. The evolution might mean adding concurrency or using in fully nonblocking environments (see [Chapter 5](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch05.html#Going-fully-reactive)). Remember that turning a nonblocking API into a blocking one is as easy as calling toBlocking(). The opposite is often challenging and requires lots of extra resources. Also, it is very difficult to predict the evolution of methods like rxBookTicket(), if they ever touch the network or filesystem, not to mention database, it is worth it to wrap them with an Observable indicating possible latency on the type level:

Observable<Ticket> rxBookTicket(Flight flight, Passenger passenger) {

//...

}

But now zipWith() returns an awkward Observable<Observable<Ticket>> and the code no longer compiles. A good rule of thumb is that whenever you see double-wrapped type (for example Optional<Optional<...>>) there is a flatMap() invocation missing somewhere. That’s the case here, as well. zipWith() takes a pair (or more generally a tuple) of events, applies a function taking these events as arguments, and puts the result into the downstream Observable as-is. This is why we saw Observable<Ticket> first but now it’s Observable<Observable<Ticket>>, where Observable<Ticket> is the result of our supplied function. There are two ways to overcome this problem. One way is by using an intermediate pair returned from zipWith:

import org.apache.commons.lang3.tuple.Pair;

Observable<Ticket> ticket = flight

.zipWith(passenger, (Flight f, Passenger p) -> Pair.of(f, p))

.flatMap(pair -> rxBookTicket(pair.getLeft(), pair.getRight()));

If using an explicit Pair from third-party library did not obscure flow enough, method reference would actually work: Pair::of, but again, we decided that visible type information is more valuable than saving a few keystrokes. After all we read code for much more time than we write it. An alternative to an intermediate pair is applying a flatMap with an identity function:

Observable<Ticket> ticket = flight

.zipWith(passenger, this::rxBookTicket)

.flatMap(obs -> obs);

This obs -> obs lambda expression is seemingly not doing anything, at least if it were a map() operator. But remember that flatMap() applies a function to each value inside Observable, so this function takes Observable<Ticket> as an argument in our case. Later, the result is not placed directly in the resulting stream, like with map(). Instead, the return value (of type Observable<T>) is “flattened,” leading to an Observable<T> rather than Observable<Observable<T>>. When dealing with schedulers, the flatMap() operator becomes even more powerful. You might perceive flatMap() as merely a syntactic trick to avoid a nested Observable<Observable<...>> problem, but it’s much more fundamental than this.

# Observable.subscribeOn() Use Cases

It is tempting to think that subscribeOn() is the right tool for concurrency in RxJava. This operator works but you should not see the usage of subscribeOn() (and yet to be described observeOn()) often. In real life, Observables come from asynchronous sources, so custom scheduling is not needed at all. We use subscribeOn() throughout this chapter to explicitly show how to upgrade existing applications to use reactive principles selectively. But in practice, Schedulers and subscribeOn() are weapons of last resort, not something seen commonly.

# flatMap() as Asynchronous Chaining Operator

In our sample application, we must now send a list of Tickets via e-mail. But we must keep in mind the following:

1. The list can be potentially quite long.
2. Sending an email might take several milliseconds or even seconds.
3. The application must keep running gracefully in case of failures, but report in the end which tickets failed to be delivered.

The last requirement quickly rules out simple tickets.forEach(this::sendEmail) because it eagerly throws an exception and won’t continue with tickets that were not yet delivered. Exceptions are actually a nasty back door to the type system and just like callbacks are not very friendly when you want to manage them in a more robust way. That is why RxJava models them explicitly as special notifications, but be patient, we will get there. In light of the error-handling requirement, our code looks more-or-less like that:

List<Ticket> failures = new ArrayList<>();

for(Ticket ticket: tickets) {

try {

sendEmail(ticket);

} catch (Exception e) {

log.warn("Failed to send {}", ticket, e);

failures.add(ticket);

}

}

However, the first two requirements or guidelines aren’t addressed. There is no reason why we keep sending emails from one thread sequentially. Traditionally, we could use an ExecutorService pool for that by submitting each email as a separate task:

List<Pair<Ticket, Future<SmtpResponse>>> tasks = tickets

.stream()

.map(ticket -> Pair.of(ticket, sendEmailAsync(ticket)))

.collect(toList());

List<Ticket> failures = tasks.stream()

.flatMap(pair -> {

try {

Future<SmtpResponse> future = pair.getRight();

future.get(1, TimeUnit.SECONDS);

return Stream.empty();

} catch (Exception e) {

Ticket ticket = pair.getLeft();

log.warn("Failed to send {}", ticket, e);

return Stream.of(ticket);

}

})

.collect(toList());

//------------------------------------

private Future<SmtpResponse> sendEmailAsync(Ticket ticket) {

return pool.submit(() -> sendEmail(ticket));

}

That’s a fair amount of code that all Java programmers should be familiar with. Yet it seems too verbose and accidentally complex. First, we iterate over tickets and submit them to a thread pool. To be precise, we call the sendEmailAsync() helper method that submits sendEmail() invocation wrapped in Callable<SmtpResponse> to a thread pool. Even more precise instances of Callable are first placed in an unbounded (by default) queue in front of a thread pool. Lack of mechanisms that slow down too rapid submission of tasks if they cannot be processed on time led to reactive streams and backpressure effort (see [“Backpressure”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch06.html#Backpressure)).

Because later we will need a Ticket instance in case of failure, we must keep track of which Future was responsible for which Ticket, again in a Pair. In real production code, you should consider a more meaningful and dedicated container like a TicketAsyncTask value object. We collect all such pairs and proceed to the next iteration. At this point, the thread pool is already running multiple sendEmail() invocations concurrently, which is precisely what we were aiming at. The second loop goes through all Futures and tries to dereference them by blocking (get()) and awaiting for completion. If get() returns successfully, we skip such a Ticket. However, if there is an exception we return Ticket instance that was associated with this task—we know it failed and we want to report it later. Stream.flatMap() allows us to return zero or one elements (or actually any number), contrary to Stream.map(), which always requires one.

You might be wondering why we need two loops instead of just one like this:

//WARNING: code is sequential despite utilizing thread pool

List<Ticket> failures = tickets

.stream()

.map(ticket -> Pair.of(ticket, sendEmailAsync(ticket)))

.flatMap(pair -> {

//...

})

.collect(toList());

This is an interesting bug that is really difficult to find if you don’t understand how Streams in Java 8 work. Because streams—just like Observables—are lazy, they evaluate the underlying collection one element at a time and only when terminal operation was requested (e.g., collect(toList())). This means that a map() operation starting background tasks is not executed on all tickets immediately; rather, it’s done one at a time, alternately by using a flatMap() operation. Furthermore, we really start one Future, block waiting for it, start a second Future, block waiting on that, and so on. An intermediate collection is needed to force evaluation, not because of clarity or readability. After all, List<Pair<Ticket, Future<SmtpResponse>>> type is hardly more readable.

That’s plenty of work and the possibility of mistake is high, so it’s no wonder that developers are reluctant to apply concurrent code on a daily basis. The little-known [ExecutorCompletionService from JDK](http://bit.ly/2d3eD4x) is sometimes used when there is a pool of asynchronous tasks and we want to process them as they complete. Moreover, Java 8 brings CompletableFuture (see [“CompletableFuture and Streams”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch05.html#CompletableFuture-interoperability)) that is entirely reactive and nonblocking. But how can RxJava help here? First, assume that an API for sending an email is already retrofitted to use RxJava:

import static rx.Observable.fromCallable;

Observable<SmtpResponse> rxSendEmail(Ticket ticket) {

//unusual synchronous Observable

return fromCallable(() -> sendEmail())

}

There is no concurrency involved, just wrapping sendEmail() inside an Observable. This is a rare Observable; ordinarily you would use subscribeOn() in the implementation so that the Observable is asynchronous by default. At this point, we can iterate over all tickets as before:

List<Ticket> failures = Observable.from(tickets)

.flatMap(ticket ->

rxSendEmail(ticket)

.flatMap(response -> Observable.<Ticket>empty())

.doOnError(e -> log.warn("Failed to send {}", ticket, e))

.onErrorReturn(err -> ticket))

.toList()

.toBlocking()

.single();

# Observable.ignoreElements()

It is easy to see that inner flatMap() in our example ignores response and returns an empty stream. In such cases, flatMap() is an overkill; the ignoreElements() operator is far more efficient. ignoreElements() simply ignores all emitted values and forwards onCompleted() or onError() notifications. Because we are ignoring the actual response and just deal with errors, ignoreElements() works great here.

All we are interested in lies inside the outer flatMap(). If it were just flatMap(this::rxSendEmail), code would work; however, any failure emitted from rxSendEmail would terminate the entire stream. But we want to “catch” all emitted errors and collect them for later consumption. We use a similar trick to Stream.flatMap(): if response was successfully emitted, we transform it to an empty Observable. This basically means that we discard successful tickets. However, in case of failures, we return a ticket that raised an exception. An extra doOnError() callback allows us to log exception—of course we can just as well add logging to onErrorReturn() operator, but I found this separation of concerns more functional.

To remain compatible with previous implementations, we transform Observable into Observable<List<Ticket>>, BlockingObservable<List<Ticket>>, toBlocking(), and finally List<Ticket> (single()). Interestingly, even BlockingObservable remains lazy. A toBlocking() operator on its own doesn’t force evaluation by subscribing to the underlying stream and it doesn’t even block. Subscription and thus iteration and sending emails is postponed until single() is invoked.

Note that if we replace the outer flatMap() with concatMap() (see [“Ways of Combining Streams: concat(), merge(), and switchOnNext()”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch03.html#Ways-of-combining-streams-concat-merge-and-switchOnNext) and [“Preserving Order Using concatMap()”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch03.html#Preserving-order-with-concatMap)), we will encounter a similar bug as the mentioned with JDK’s Stream. As opposed to flatMap() (or merge) that subscribe immediately to all inner streams, concatMap (or concat) subscribes one inner Observable after another. And as long as no one subscribed to Observable, no work even began.

So far, a simple for loop with a try—catch was replaced with less readable and more complex Observable. However, to turn our sequential code into multithreaded computation we barely need to add one extra operator:

Observable

.from(tickets)

.flatMap(ticket ->

rxSendEmail(ticket)

.ignoreElements()

.doOnError(e -> log.warn("Failed to send {}", ticket, e))

.onErrorReturn(err -> ticket)

.subscribeOn(Schedulers.io()))

It is so noninvasive that you might find it hard to spot. One extra subscribeOn() operator causes each individual rxSendMail() to be executed on a specified Scheduler (io(), in this case). This is one of the strengths of RxJava; it is not opinionated about threading, defaulting to synchronous execution but allowing seamless and almost transparent multithreading. Of course, this doesn’t mean that you can safely inject schedulers in arbitrary places. But at least the API is less verbose and higher level. We will explore schedulers in much more detail later in [“Multithreading in RxJava”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#threads-in-rxjava). For the time being remember that Observables are synchronous by default; however, we can easily change that and apply concurrency in places where it was least expected. This is especially valuable in existing legacy applications, which you can optimize without much hassle.

Wrapping up if you are implementing Observables from scratch, making them asynchronous by default is more idiomatic. That means placing subscribeOn() directly inside rxSendEmail() rather than externally. Otherwise, you risk wrapping already asynchronous streams with yet another layer of schedulers. Of course, if the producer behind Observable is already asynchronous, it is even better because your stream does not bind to any particular thread. Additionally, you should postpone subscribing to an Observable as late as possible, typically close to the web framework of our outside world. This changes your mindset significantly. Your entire business logic is lazy until someone actually wants to see the results.[3](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#idm140239471172304)

# Replacing Callbacks with Streams

Traditional APIs are blocking most of the time, meaning they force you to wait synchronously for the results. This approach works relatively well, at least before you heard about RxJava. But a blocking API is particularly problematic when data needs to be pushed from the API producer to consumers—this is anarea where RxJava really shines. There are numerous examples of such cases and various approaches are taken by API designers. Typically, we need to provide some sort of a callback that the API invokes, often called event listeners. One of the most common scenarios like that is [Java Message Service (JMS)](http://bit.ly/2d3hx9m). Consuming JMS typically involves implementing a class that the application server or container notifies on every incoming messages. We can replace with relative ease such listeners with a composable Observable, which is much more robust and versatile. The traditional listener looks similar to this class, here using JMS support in [Spring framework](http://bit.ly/2d3hieL), but our solution is technology-agnostic:

@Component

class JmsConsumer {

@JmsListener(destination = "orders")

public void newOrder(Message message) {

//...

}

}

When a JMS message is received, the JmsConsumer class must decide what to do with it. Typically, some business logic is invoked inside a message consumer. When a new component wants to be notified about such messages, it must modify JmsConsumer appropriately. Coversely, imagine Observable<Message> that can be subscribed to by anyone. Moreover, an entire universe of RxJava operators is available, allowing mapping, filtering, and combining capabilities. The easiest way to convert from a push, callback-based API to Observable is to use Subjects. Every time a new JMS message is delivered, we push that message to a PublishSubject that looks like an ordinary hot Observable from the outside:

private final PublishSubject<Message> subject = PublishSubject.create();

@JmsListener(destination = "orders", concurrency="1")

public void newOrder(Message msg) {

subject.onNext(msg);

}

Observable<Message> observe() {

return subject;

}

Keep in mind that Observable<Message> is hot; it begins emitting JMS messages as soon as they are consumed. If no one is subscribed at the moment, messages are simply lost. ReplaySubject is an alternative, but because it caches all events since the application startup, it’s not suitable for long-running processes. In case you have a subscriber that absolutely must receive all messages, ensure that it subscribes before the JMS message listener is initialized. Additionally, our message listener has a concurrency="1" parameter to ensure that Subject is not invoked from multiple threads. As an alternative, you can use Subject.toSerialized().

As a side note, Subjects are easier to get started but are known to be problematic after a while. In this particular case, we can easily replace Subject with the more idiomatic RxJava Observable that uses create() directly:

public Observable<Message> observe(

ConnectionFactory connectionFactory,

Topic topic) {

return Observable.create(subscriber -> {

try {

subscribeThrowing(subscriber, connectionFactory, topic);

} catch (JMSException e) {

subscriber.onError(e);

}

});

}

private void subscribeThrowing(

Subscriber<? super Message> subscriber,

ConnectionFactory connectionFactory,

Topic orders) throws JMSException {

Connection connection = connectionFactory.createConnection();

Session session = connection.createSession(true, AUTO\_ACKNOWLEDGE);

MessageConsumer consumer = session.createConsumer(orders);

consumer.setMessageListener(subscriber::onNext);

subscriber.add(onUnsubscribe(connection));

connection.start();

}

private Subscription onUnsubscribe(Connection connection) {

return Subscriptions.create(() -> {

try {

connection.close();

} catch (Exception e) {

log.error("Can't close", e);

}

});

}

The JMS API provides two ways of receiving messages from a broker: synchronous via blocking receive() method, and nonblocking, using MessageListener. The nonblocking API is beneficial for many reasons; for example, it holds less resources like threads and stack memory. Also it aligns beautifully with the Rx style of programming. Rather than creating a MessageListener instance and calling our subscriber from within it, we can use this terse syntax with method reference:

consumer.setMessageListener(subscriber::onNext)

Also, we must take care of resource cleanup and proper error handling. This tiny transformation layer allows us to easily consume JMS messages without worrying about API internals. Here an example using the popular [ActiveMQ](http://activemq.apache.org) messaging broker running locally:

import org.apache.activemq.ActiveMQConnectionFactory;

import org.apache.activemq.command.ActiveMQTopic;

ConnectionFactory connectionFactory =

new ActiveMQConnectionFactory("tcp://localhost:61616");

Observable<String> txtMessages =

observe(connectionFactory, new ActiveMQTopic("orders"))

.cast(TextMessage.class)

.flatMap(m -> {

try {

return Observable.just(m.getText());

} catch (JMSException e) {

return Observable.error(e);

}

});

JMS, just like JDBC, has a reputation of heavily using checked JMSException, even when calling getText() on a TextMessage. To properly handle errors (see [“Error Handling”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch07.html#Error-Handling) for more details) we use flatMap() and wrap exceptions. From that point, you can treat JMS messages flowing in like any other asynchronous and nonblocking stream. And by the way, we used the cast() operator that optimistically casts upstream events to a given type, failing with onError(), otherwise. cast() is basically a specialized map() operator that behaves like map(x -> (TextMessage)x).

# Polling Periodically for Changes

The worst blocking API that you can work with requires polling for changes. It provides no mechanism to push changes right at you, even with callbacks or by blocking indefinitely. The only mechanism this API gives is asking for the current state, and it is up to you to figure out if it differs from previous state or not. RxJava has few really powerful operators that you can apply to retrofit a given API to Rx style. The first case I want you to consider is a simple method that delivers a single value that represents state, for example long getOrderBookLength(). To track changes we must call this method frequently enough and capture differences. You can achieve this in RxJava with a very basic operator composition:

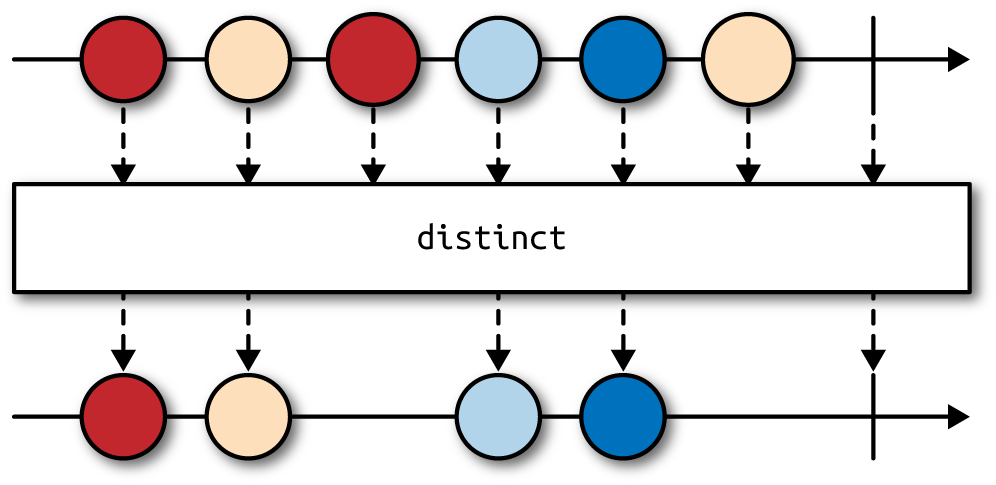
Observable

.interval(10, TimeUnit.MILLISECONDS)

.map(x -> getOrderBookLength())

.distinctUntilChanged()

First we produce a synthetic long value every 10 milliseconds which serves as a basic ticking counter. For each such value (that is every 10 milliseconds), we call getOrderBookLength(). However, the aforementioned method doesn’t change that often, and we don’t want to flood our subscribers with lots of irrelevant state changes. Luckily we can simply say distinctUntilChanged() and RxJava will transparently skip long values returned by getOrderBookLength() that have not changed since last invocation, as demonstrated in the following marble diagram:



We can apply this pattern even further. Imagine that you are watching for filesystem or database table changes. The only mechanism at your disposal is taking a current snapshot of files or database records. You are building an API that will notify clients about every new item. Obviously, you can use java.nio.file.WatchService or database triggers, but take this as an educational example. This time, again, we begin by periodically taking a snapshot of current state:

Observable<Item> observeNewItems() {

return Observable

.interval(1, TimeUnit.SECONDS)

.flatMapIterable(x -> query())

.distinct();

}

List<Item> query() {

//take snapshot of file system directory

//or database table

}

The distinct() operator keeps a record of all items that passed through it (see also [“Dropping Duplicates Using distinct() and distinctUntilChanged()”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch03.html#Dropping-duplicates-with-distinct-and-distinctUntilChanged)). If the same item appears for the second time, it is simply ignored. That is why we can push the same list of Items every second. The first time they are pushed downstream to all subscribers. However, when the exact same list appears one second later, all items were already seen and are therefore discarded. If at some point in time the list returned from query() contains one extra Item, distinct() will let it go but discard it the next time. This simple pattern allows us to replace a bunch of Thread.sleep() invocations and manual caching with periodic polling. It is applicable in many areas, like [File Transfer Protocol (FTP)](https://en.wikipedia.org/wiki/File_Transfer_Protocol) polling, web scraping, and so on.

# Multithreading in RxJava

There are third-party APIs that are blocking and there is simply nothing we can do about it. We might not have source code, rewriting might result in too much risk. In that case, we must learn how to deal with blocking code rather than fighting it.

One of the hallmarks of RxJava is declarative concurrency, as opposed to imperative concurrency. Manually creating and managing threads is a thing of the past (compare with [“Thread Pool of Connections”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/app01.html#Thread-pool-of-connections)) most of us already use managed thread pools (e.g., with ExecutorService). But RxJava goes one step further: Observable can be nonblocking just like CompletableFuture in Java 8 (see [“CompletableFuture and Streams”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch05.html#CompletableFuture-interoperability)), but unlike the other, it is also lazy. Unless you subscribe, a well-behaving Observable will not perform any action. But the power of Observable goes even beyond that.

An asynchronous Observable is the one that calls your Subscribers callback methods (like onNext()) from a different thread. Recall [“Mastering Observable.create()”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch02.html#Mastering-observable-create) in which we explored when subscribe() is blocking, waiting until all notifications arrive? In real life, most Observables come from sources that are asynchronous by their nature. [Chapter 5](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch05.html#Going-fully-reactive) is entirely devoted to such Observables. But even our simple JMS example from [“Replacing Callbacks with Streams”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#replacing-callbacks-with-streams), which uses a built-in, nonblocking API from the JMS specification (MessageListener interface). This is not enforced or suggested by the type system, but many Observables are asynchronous from the very beginning, and you should assume that. A blocking subscribe() method happens very rarely, when a lambda within Observable.create() is not backed by any asynchronous process or stream. However, by default (with create()) everything happens in the client thread (the one that subscribed). If you just poke onNext() directly within your create() callback, no multithreading and concurrency is involved whatsoever.

Encountering such an unusual Observable, we can declaratively select the so-called Scheduler that will be used to emit values. In case of CompletableFuture, we have no control over underlying threads, the API made the decision and in worst case it is impossible to override it. RxJava rarely makes such decisions alone and chooses safe default: client thread and no multithreading involved. For the purposes of this chapter, we will use a really simple logging “library,”[4](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#idm140239470771152) which will print a message along with the current thread and number of milliseconds since the start of the program using System.currentTimeMillis():

void log(Object label) {

System.out.println(

System.currentTimeMillis() - start + "\t| " +

Thread.currentThread().getName() + "\t| " +

label);

}

## What Is a Scheduler?

RxJava is concurrency-agnostic, and as a matter of fact it does not introduce concurrency on its own. However, some abstractions to deal with threads are exposed to the end user. Also, certain operators cannot work properly without concurrency; see [“Other Uses for Schedulers”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#Other-usages-of-Schedulers) for some of them. Luckily, the Scheduler class, the only one you must pay attention to, is fairly simple. In principle it works similarly to ScheduledExecutorService from java.util.concurrent—it executes arbitrary blocks of code, possibly in the future. However, to meet Rx contract, it offers some more fine-grained abstractions, which you can see more of in the advanced section [“Scheduler implementation details overview”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#Scheduler-implementation-details-overview).

Schedulers are used together with subscribeOn() and observeOn() operators as well as when creating certain types of Observables. A scheduler only creates instances of Workers that are responsible for scheduling and running code. When RxJava needs to schedule some code it first asks Scheduler to provide a Worker and uses the latter to schedule subsequent tasks. You will find examples of this API later on, but first familiarize yourself with available built-in schedulers:

Schedulers.newThread()

This scheduler simply starts a new thread every time it is requested via subscribeOn() or observeOn(). newThread() is hardly ever a good choice, not only because of the latency involved when starting a thread, but also because this thread is not reused. Stack space must be allocated up front (typically around one megabyte, as controlled by the -Xss parameter of the JVM) and the operating system must start new native thread. When the Worker is done, the thread simply terminates. This scheduler can be useful only when tasks are coarse-grained: it takes a lot of time to complete but there are very few of them, so that threads are unlikely to be reused at all. See also: [“Thread per Connection”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/app01.html#Thread-per-connection). In practice, following Schedulers.io() is almost always a better choice.

Schedulers.io()

This scheduler is similar to newThread(), but already started threads are recycled and can possibly handle future requests. This implementation works similarly to ThreadPoolExecutor from java.util.concurrent with an unbounded pool of threads. Every time a new Worker is requested, either a new thread is started (and later kept idle for some time) or the idle one is reused.

The name io() is not a coincidence. Consider using this scheduler for I/O bound tasks which require very little CPU resources. However they tend to take quite some time, waiting for network or disk. Thus, it is a good idea to have a relatively big pool of threads. Still, be careful with unbounded resources of any kind—in case of slow or unresponsive external dependencies like web services, io() scheduler might start an enormous number of threads, leading to your very own application becoming unresponsive, as well. See [“Managing Failures with Hystrix”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch08.html#Hystrix) for more details how to tackle this problem.

Schedulers.computation()

You should use a computation scheduler when tasks are entirely CPU-bound; that is, they require computational power and have no blocking code (reading from disk, network, sleeping, waiting for lock, etc.) Because each task executed on this scheduler is supposed to fully utilize one CPU core, executing more such tasks in parallel than there are available cores would not bring much value. Therefore, computation() scheduler by default limits the number of threads running in parallel to the value of availableProcessors(), as found in the Runtime.getRuntime() utility class.

If for some reason you need a different number of threads than the default, you can always use the rx.scheduler.max-computation-threads system property. By taking less threads you ensure that there is always one or more CPU cores idle, and even under heavy load, computation() thread pool does not saturate your server. It is not possible to have more computation threads than cores.

computation() scheduler uses unbounded queue in front of every thread, so if the task is scheduled but all cores are occupied, they are queued. In case of load peak, this scheduler will keep the number of threads limited. However, the queue just before each thread will keep growing.

Luckily, built-in operators, especially observeOn() that we are about to discover in [“Declarative Concurrency with observeOn()”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#Declarative-concurrency-with-observeOn) ensure that this Scheduler is not overloaded.

Schedulers.from(Executor executor)

Schedulers are internally more complex than Executors from java.util.concurrent, so a separate abstraction was needed. But because they are conceptually quite similar, unsurprisingly there is a wrapper that can turn Executor into Scheduler using the from() factory method:

import com.google.common.util.concurrent.ThreadFactoryBuilder;

import rx.Scheduler;

import rx.schedulers.Schedulers;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.LinkedBlockingQueue;

import java.util.concurrent.ThreadFactory;

import java.util.concurrent.ThreadPoolExecutor;

//...

ThreadFactory threadFactory = new ThreadFactoryBuilder()

.setNameFormat("MyPool-%d")

.build();

Executor executor = new ThreadPoolExecutor(

10, //corePoolSize

10, //maximumPoolSize

0L, TimeUnit.MILLISECONDS, //keepAliveTime, unit

new LinkedBlockingQueue<>(1000), //workQueue

threadFactory

);

Scheduler scheduler = Schedulers.from(executor);

I am intentionally using this verbose syntax for creating ExecutorService rather than the more simple version:

import java.util.concurrent.Executors;

//...

ExecutorService executor = Executors.newFixedThreadPool(10);

Although tempting, the Executors factory class hardcodes several defaults that are impractical or even dangerous in enterprise applications. For examples, it uses unbounded LinkedBlockingQueue that can grow infinitely, resulting in OutOfMemoryError for cases in which there are a of large number of outstanding tasks. Also, the default ThreadFactory uses meaningless thread names like pool-5-thread-3. Naming threads properly is an invaluable tool when profiling or analyzing thread dumps. Implementing ThreadFactory from scratch is a bit cumbersome, so we used ThreadFactoryBuilder from [Guava](https://github.com/google/guava). If you are interested in tuning and properly utilizing thread pools even further, see [“Thread Pool of Connections”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/app01.html#Thread-pool-of-connections) and [“Managing Failures with Hystrix”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch08.html#Hystrix). Creating schedulers from Executor that we consciously configured is advised for projects dealing with high load. However, because RxJava has no control over independently created threads in an Executor, it cannot pin threads (that is, try to keep work of the same task on the same thread to improve cache locality). This Scheduler barely makes sure a single Scheduler.Worker (see [“Scheduler implementation details overview”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#Scheduler-implementation-details-overview)) processes events sequentially.

Schedulers.immediate()

Schedulers.immediate() is a special scheduler that invokes a task within the client thread in a blocking fashion, rather than asynchronously. Using it is pointless unless some part of your API requires providing a scheduler, whereas you are absolutely fine with default behavior of Observable, not involving any threading at all. In fact, subscribing to an Observable (more on that in a second) via immediate() Scheduler typically has the same effect as not subscribing with any particular scheduler at all. In general, avoid this scheduler, it blocks the calling thread and is of limited use.

Schedulers.trampoline()

The trampoline() scheduler is very similar to immediate() because it also schedules tasks in the same thread, effectively blocking. However, as opposed to immediate(), the upcoming task is executed when all previously scheduled tasks complete. immediate() invokes a given task right away, whereas trampoline() waits for the current task to finish. Trampoline is a pattern in functional programming that allows implementing recursion without infinitely growing the call stack. This is best explained with an example, first involving immediate(). By the way, notice that we do not interact directly with a Scheduler instance but first create a Worker. This makes sense as you will quickly see in [“Scheduler implementation details overview”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#Scheduler-implementation-details-overview).

Scheduler scheduler = Schedulers.immediate();

Scheduler.Worker worker = scheduler.createWorker();

log("Main start");

worker.schedule(() -> {

log(" Outer start");

sleepOneSecond();

worker.schedule(() -> {

log(" Inner start");

sleepOneSecond();

log(" Inner end");

});

log(" Outer end");

});

log("Main end");

worker.unsubscribe();

The output is as expected; you could actually replace schedule() with a simple method invocation:

1044 | main | Main start

1094 | main | Outer start

2097 | main | Inner start

3097 | main | Inner end

3100 | main | Outer end

3100 | main | Main end

Inside the Outer block we schedule() Inner block that gets invoked immediately, interrupting the Outer task. When Inner is done, the control goes back to Outer. Again, this is simply a convoluted way of invoking a task in a blocking manner indirectly via immediate() Scheduler. But what happens if we replace Schedulers.immediate() with Schedulers.trampoline()? The output is quite different:

1030 | main | Main start

1096 | main | Outer start

2101 | main | Outer end

2101 | main | Inner start

3101 | main | Inner end

3101 | main | Main end

Do you see how Outer manages to complete before Inner even starts? This is because the Inner task was queued inside the trampoline() Scheduler, which was already occupied by the Outer task. When Outer finished, the first task from the queue (Inner) began. We can go even further to make sure you understand the difference:

log("Main start");

worker.schedule(() -> {

log(" Outer start");

sleepOneSecond();

worker.schedule(() -> {

log(" Middle start");

sleepOneSecond();

worker.schedule(() -> {

log(" Inner start");

sleepOneSecond();

log(" Inner end");

});

log(" Middle end");

});

log(" Outer end");

});

log("Main end");

The Worker from immediate() Scheduler outputs the following:

1029 | main | Main start

1091 | main | Outer start

2093 | main | Middle start

3095 | main | Inner start

4096 | main | Inner end

4099 | main | Middle end

4099 | main | Outer end

4099 | main | Main end

Versus the trampoline() worker:

1041 | main | Main start

1095 | main | Outer start

2099 | main | Outer end

2099 | main | Middle start

3101 | main | Middle end

3101 | main | Inner start

4102 | main | Inner end

4102 | main | Main end

Schedulers.test()

This Scheduler is used only for testing purposes, and you will never see it in production code. Its main advantage is the ability to arbitrarily advance the clock, simulating time passing by. TestScheduler is described to a great extent in [“Schedulers in Unit Testing”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch07.html#Dealing-with-schedulers). Schedulers alone are not very interesting. If you want to discover how they work internally and how to implement your own, check out the next section.

### Scheduler implementation details overview

###### Note

This section is entirely optional, feel free to jump straight to [“Declarative Subscription with subscribeOn()”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#declarative-subscription-with-subscribeOn) if you are not interested in implementation details.

Scheduler not only decouples tasks and their execution (typically by running them in another thread), but it also abstracts away the clock, as we will learn in [“Virtual Time”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch07.html#Virtual-Time). The API of the Scheduler is a bit simpler compared to, for example, ScheduledExecutorService:

abstract class Scheduler {

abstract Worker createWorker();

long now();

abstract static class Worker implements Subscription {

abstract Subscription schedule(Action0 action);

abstract Subscription schedule(Action0 action,

long delayTime, TimeUnit unit);

long now();

}

}

When RxJava wants to schedule a task (presumably, but not necessarily in the background), it must first ask for an instance of Worker. It is the Worker that allows scheduling the task without any delay or at some point in time. Both Scheduler and Worker have an overridable source of time (now() method) that it uses to determine when a given task is supposed to run. Naively, you can think of a Scheduler like a thread pool and a Worker like a thread inside that pool.

The separation between Scheduler and Worker is necessary to easily implement some of the guidelines enforced by the Rx contract, namely invoking Subscriber’s method sequentially, not concurrently. Worker’s contract provides just that: two tasks scheduled on the same Worker will never run concurrently. However, independent Workers from the same Scheduler can run tasks concurrently just fine.

Rather than going through the API, let’s analyze the source code of an existing Scheduler, namely HandlerScheduler, as found in the [RxAndroid project](https://github.com/ReactiveX/RxAndroid). This Scheduler simply runs all scheduled tasks on an Android UI thread. Updating the user interface is only allowed from that thread (see [“Android Development with RxJava”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch08.html#android) for more details). This is similar to the [Event Dispatch Thread (EDT)](http://bit.ly/2cMxH4U) as found in Swing, where most of the updates to windows and components must be executed within dedicated thread (EDT). Unsurprisingly, there is also the RxSwing[5](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#idm140239470096960) project for that.

The code snippet that follows is a stripped down and incomplete class from RxAndroid for education purposes only:

package rx.android.schedulers;

import android.os.Handler;

import android.os.Looper;

import rx.Scheduler;

import rx.Subscription;

import rx.functions.Action0;

import rx.internal.schedulers.ScheduledAction;

import rx.subscriptions.Subscriptions;

import java.util.concurrent.TimeUnit;

public final class SimplifiedHandlerScheduler extends Scheduler {

@Override

public Worker createWorker() {

return new HandlerWorker();

}

static class HandlerWorker extends Worker {

private final Handler handler = new Handler(Looper.getMainLooper());

@Override

public void unsubscribe() {

//Implementation coming soon...

}

@Override

public boolean isUnsubscribed() {

//Implementation coming soon...

return false;

}

@Override

public Subscription schedule(final Action0 action) {

return schedule(action, 0, TimeUnit.MILLISECONDS);

}

@Override

public Subscription schedule(

Action0 action, long delayTime, TimeUnit unit) {

ScheduledAction scheduledAction = new ScheduledAction(action);

handler.postDelayed(scheduledAction, unit.toMillis(delayTime));

scheduledAction.add(Subscriptions.create(() ->

handler.removeCallbacks(scheduledAction)));

return scheduledAction;

}

}

}

Details of the Android API are not important at the moment. What happens here is that every time we schedule something on a HandlerWorker, the block of code is passed to a special postDelayed() method that executes it on a dedicated Android thread. There is just one such thread, so events are serialized not only within, but also across Workers.

Before we pass action to be executed, we wrap it with ScheduledAction, which implements both Runnable and Subscription. RxJava is lazy whenever it can be—this also applies to scheduling tasks. If for any reason you decide that a given action should not be executed after all (this makes sense when the action was scheduled in the future, not immediately), simply run unsubscribe() on the Subscription returned from schedule(). It is the responsibility of the Worker to properly handle unsubscription (best effort at least).

Client code can also decide to unsubscribe() from Worker in its entirety. This should unsubscribe all queued tasks as well as release the Worker so that the underlying thread can potentially be reused later. The following code snippet enhances the SimplifiedHandlerScheduler by adding Worker unsubscription flow (only modified methods are included):

private CompositeSubscription compositeSubscription =

new CompositeSubscription();

@Override

public void unsubscribe() {

compositeSubscription.unsubscribe();

}

@Override

public boolean isUnsubscribed() {

return compositeSubscription.isUnsubscribed();

}

@Override

public Subscription schedule(Action0 action, long delayTime, TimeUnit unit) {

if (compositeSubscription.isUnsubscribed()) {

return Subscriptions.unsubscribed();

}

final ScheduledAction scheduledAction = new ScheduledAction(action);

scheduledAction.addParent(compositeSubscription);

compositeSubscription.add(scheduledAction);

handler.postDelayed(scheduledAction, unit.toMillis(delayTime));

scheduledAction.add(Subscriptions.create(() ->

handler.removeCallbacks(scheduledAction)));

return scheduledAction;

}

In [“Controlling Listeners by Using Subscription and Subscriber<T>”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch02.html#controlling-listeners-with-subscription-and-subscribert), we explored the Subscription interface but never really looked at the implementation details. CompositeSubscription is one out of many implementations available that itself is just a container for child Subscriptions (a Composite design pattern). Unsubscribing from CompositeSubscription means unsubscribing from all children. You also can add and remove the children managed by CompositeSubscription.

In our custom Scheduler, CompositeSubscription is used to track all Subscriptions from the previous schedule() invocations (see compositeSubscription.add(scheduledAction)). On the other hand, the child ScheduledAction needs to know about its parent (see: addParent()) so that it can remove itself when the action is completed or canceled. Otherwise, Worker would accumulate stale child Subscriptions forever. When the client code decides that it no longer needs a HandlerWorker instance, it unsubscribes from it. The unsubscription is propagated to all (if any) outstanding child Subscriptions.

That was a very brief introduction to Schedulers in RxJava. The details of their internals are not that useful in daily work; as a matter of fac, they are designed in such as way as to make using RxJava more intuitive and predictable. That being said, let’s quickly see how Schedulers solve many concurrency problems in Rx.

## Declarative Subscription with subscribeOn()

In [“Mastering Observable.create()”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch02.html#Mastering-observable-create) we saw that subscribe() by default uses the client thread. To recap, here is the most simple subscription that you can come up with where no threading was involved whatsoever:

Observable<String> simple() {

return Observable.create(subscriber -> {

log("Subscribed");

subscriber.onNext("A");

subscriber.onNext("B");

subscriber.onCompleted();

});

}

//...

log("Starting");

final Observable<String> obs = simple();

log("Created");

final Observable<String> obs2 = obs

.map(x -> x)

.filter(x -> true);

log("Transformed");

obs2.subscribe(

x -> log("Got " + x),

Throwable::printStackTrace,

() -> log("Completed")

);

log("Exiting");

Notice where the logging statements are placed and study the output carefully, especially with regard to which thread invoked the print statement:

33 | main | Starting

120 | main | Created

128 | main | Transformed

133 | main | Subscribed

133 | main | Got A

133 | main | Got B

133 | main | Completed

134 | main | Exiting

Pay attention: the order of statements is absolutely predictable. First, every line of code in the preceding code snippet runs in the main thread, there are no thread pools and no asynchronous emission of events involved. Second, the order of execution might not be entirely clear at first sight.

When the program starts, it prints Starting, which is understandable. After creating an instance of Observable<String>, we see the Created message. Notice that Subscribed appears later, when we actually subscribe. Without the subscribe() invocation, the block of code inside Observable.create() is never executed. Moreover, even map() and filter() operators do not have any visible side effects, notice how the Transformed message is printed even before Subscribed.

Later, we receive all emitted events and completion notification. Finally, the Exiting statement is printed and the program can return. This is an interesting observation—subscribe() was supposed to be registering a callback when events appear asynchronously. This is the assumption that you should make by default. However in this case there is no threading involved and subscribe() is actually blocking. How is this so?

There is an inherent but hidden connection between subscribe() and create(). Every time you call subscribe() on an Observable, its OnSubscribe callback method is invoked (wrapping the lambda expression you passed to create()). It receives your Subscriber as an argument. By default, this happens in the same thread and is blocking, so whatever you do inside create() will block subscribe(). If your create() method sleeps for few seconds, subscribe() will block. Moreover, if there are operators between Observable.create() and your Subscriber (lambda acting as callback), all these operators are invoked on behalf of the thread that invoked subscribe(). RxJava does not inject any concurrency facilities by default between Observable and Subscriber. The reason behind that is that Observables tend to be backed by other concurrency mechanisms like event loops or custom threads, so Rx lets you take full control rather than imposing any convention.

This observation prepares the landscape for the subscribeOn() operator. By inserting subscribeOn() anywhere between an original Observable and subscribe(), you declaratively select Scheduler where the OnSubscribe callback method will be invoked. No matter what you do inside create(), this work is offloaded to an independent Scheduler and your subscribe() invocation no longer blocks:

log("Starting");

final Observable<String> obs = simple();

log("Created");

obs

.subscribeOn(schedulerA)

.subscribe(

x -> log("Got " + x),

Throwable::printStackTrace,

() -> log("Completed")

);

log("Exiting");

35 | main | Starting

112 | main | Created

123 | main | Exiting

123 | Sched-A-0 | Subscribed

124 | Sched-A-0 | Got A

124 | Sched-A-0 | Got B

124 | Sched-A-0 | Completed

Do you see how the main thread exits before Observable even begins emitting any values? Technically, the order of log messages is no longer that predictable because two threads are running concurrently: main, which subscribed and wants to exit, and Sched-A-0, which emits events as soon as someone subscribed. The schedulerA as well as Sched-A-0 thread come from the following sample schedulers we built for illustration purposes:

import static java.util.concurrent.Executors.newFixedThreadPool;

ExecutorService poolA = newFixedThreadPool(10, threadFactory("Sched-A-%d"));

Scheduler schedulerA = Schedulers.from(poolA);

ExecutorService poolB = newFixedThreadPool(10, threadFactory("Sched-B-%d"));

Scheduler schedulerB = Schedulers.from(poolB);

ExecutorService poolC = newFixedThreadPool(10, threadFactory("Sched-C-%d"));

Scheduler schedulerC = Schedulers.from(poolC);

private ThreadFactory threadFactory(String pattern) {

return new ThreadFactoryBuilder()

.setNameFormat(pattern)

.build();

}

These schedulers will be used across all examples, but they are fairly easy to remember. Three independent schedulers, each managing 10 threads from an ExecutorService. To make the output nicer, each thread pool has a distinct naming pattern.

Before we begin, you must understand that in mature applications, in terms of Rx adoption, subscribeOn() is very seldom used. Normally, Observables come from sources that are naturally asynchronous (like RxNetty, see [“Nonblocking HTTP Server with Netty and RxNetty”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch05.html#Non-blocking-HTTP-server-with-Netty)) or apply scheduling on their own (like Hystrix, see [“Managing Failures with Hystrix”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch08.html#Hystrix)). You should treat subscribeOn() only in special cases when the underlying Observable is known to be synchronous (create() being blocking). However, subscribeOn() is still a much better solution than hand-crafted threading within create():

//Don't do this

Observable<String> obs = Observable.create(subscriber -> {

log("Subscribed");

Runnable code = () -> {

subscriber.onNext("A");

subscriber.onNext("B");

subscriber.onCompleted();

};

new Thread(code, "Async").start();

});

The preceding code mixes two concepts: producing events and choosing concurrency strategy. Observable should be responsible only for production logic, whereas it is only the client code that can make judicious decision about concurrency. Remember that Observable is lazy but also immutable, in the sense that subscribeOn() affects only downstream subscribers, if someone subscribes to the exact same Observable without subscribeOn() in between, no concurrency will be involved by default.

Keep in mind that in this chapter our focus is on existing applications and introducing RxJava gradually. The subscribeOn() operator is quite useful in such circumstances; however, after you grasp reactive extensions and begin using them on large scale, the value of subscribeOn() diminishes. In entirely reactive software stacks, as found for example at Netflix , subscribeOn() is almost never used, yet all Observables are asynchronous. Most of the time Observables come from asynchronous sources and they are treated as asynchronous by default. Therefore, using subscribeOn() is very limited, mostly when retrofitting existing APIs or libraries. In [Chapter 5](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch05.html#Going-fully-reactive), we write write truly asynchronous applications without explicit subscribeOn() and Schedulers altogether.

## subscribeOn() Concurrency and Behavior

There are several nuances regarding how subscribeOn() works. First, curious reader should be wondering what happens if two invocations of the subscribeOn() appear between Observable and subscribe(). The answer is simple: subscribeOn() closest to the original Observable wins. This has important practical implications. If you are designing an API and you use subscribeOn() internally, the client code has no way of overriding the Scheduler of your choice. This can be a conscious design decision; after all, the API designer might know best which Scheduler is appropriate. On the other hand, providing an overloaded version of said API that allows overriding the chosen Scheduler is always a good idea.

Let’s study how subscribeOn() behaves:

log("Starting");

Observable<String> obs = simple();

log("Created");

obs

.subscribeOn(schedulerA)

//many other operators

.subscribeOn(schedulerB)

.subscribe(

x -> log("Got " + x),

Throwable::printStackTrace,

() -> log("Completed")

);

log("Exiting");

The output reveals only schedulerA’s threads:

17 | main | Starting

73 | main | Created

83 | main | Exiting

84 | Sched-A-0 | Subscribed

84 | Sched-A-0 | Got A

84 | Sched-A-0 | Got B

84 | Sched-A-0 | Completed

Interestingly, subscribing on schedulerB is not entirely ignored in favor of schedulerA. schedulerB is still used for a short period of time, but it barely schedules new action on schedulerA, which does all the work. Thus, multiple subscribeOn() are not only ignored, but also introduce small overhead.

Speaking of operators, we said that the create() method used when there is a new Subscriber is executed within the provided scheduler (if any). But which thread executes all these transformations happening between create() and subscribe()? We already know that when all operators are executed by default in the same thread (scheduler), no concurrency is involved by default:

log("Starting");

final Observable<String> obs = simple();

log("Created");

obs

.doOnNext(this::log)

.map(x -> x + '1')

.doOnNext(this::log)

.map(x -> x + '2')

.subscribeOn(schedulerA)

.doOnNext(this::log)

.subscribe(

x -> log("Got " + x),

Throwable::printStackTrace,

() -> log("Completed")

);

log("Exiting");

We sprinkled the pipeline of operators occasionally with doOnNext() to see which thread is in control at this point. Remember that the position of subscribeOn() is not relevant, it can be right after Observable or just before subscribe(). The output is unsurprising:

20 | main | Starting

104 | main | Created

123 | main | Exiting

124 | Sched-A-0 | Subscribed

124 | Sched-A-0 | A

124 | Sched-A-0 | A1

124 | Sched-A-0 | A12

124 | Sched-A-0 | Got A12

124 | Sched-A-0 | B

124 | Sched-A-0 | B1

124 | Sched-A-0 | B12

125 | Sched-A-0 | Got B12

Watch how create() is invoked and produces A and B events. These events travel sequentially through the scheduler’s thread to finally reach the Subscriber. Many newcomers to RxJava believe that using a Scheduler with a large number of threads will automatically fork processing of events concurrently and somehow join all the results together in the end. This is not the case. RxJava creates a single Worker instance (see: [“Scheduler implementation details overview”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#Scheduler-implementation-details-overview)) for the entire pipeline, mostly to guarantee sequential processing of events.

This means that if one of your operators is particularly slow—for example, map() reading data from disk in order to transform events passing by—this costly operation will be invoked within the same thread. A single broken operator can slow down the entire pipeline, from production to consumption. This is an antipattern in RxJava, operators should be nonblocking, fast, and as pure as possible.

Again, flatMap() comes to the rescue. Rather than blocking within map(), we can invoke flatMap() and asynchronously collect all the results. Therefore, flatMap() and merge() are the operators when we want to achieve true parallelism. But even with flatMap() it is not obvious. Imagine a grocery store (let’s call it “RxGroceries”) that provides an API for purchasing goods:

class RxGroceries {

Observable<BigDecimal> purchase(String productName, int quantity) {

return Observable.fromCallable(() ->

doPurchase(productName, quantity));

}

BigDecimal doPurchase(String productName, int quantity) {

log("Purchasing " + quantity + " " + productName);

//real logic here

log("Done " + quantity + " " + productName);

return priceForProduct;

}

}

Obviously, the implementation of doPurchase() is irrelevant here, just imagine it takes some time and resources to complete. We simulate business logic by adding artificial sleep of one second, slightly higher if quantity is bigger. Blocking Observables like the one returned from purchase() are unusual in a real application, but let’s keep it this way for educational purposes. When purchasing several goods we would like to parallelize as much as possible and calculate total price for all goods in the end. The first attempt is fruitless:

Observable<BigDecimal> totalPrice = Observable

.just("bread", "butter", "milk", "tomato", "cheese")

.subscribeOn(schedulerA) //BROKEN!!!

.map(prod -> rxGroceries.doPurchase(prod, 1))

.reduce(BigDecimal::add)

.single();

The result is correct, it is an Observable with just a single value: total price, calculated using reduce(). For each product, we invoke doPurchase() with quantity one. However, despite using schedulerA backed by a thread pool of 10, the code is entirely sequential:

144 | Sched-A-0 | Purchasing 1 bread

1144 | Sched-A-0 | Done 1 bread

1146 | Sched-A-0 | Purchasing 1 butter

2146 | Sched-A-0 | Done 1 butter

2146 | Sched-A-0 | Purchasing 1 milk

3147 | Sched-A-0 | Done 1 milk

3147 | Sched-A-0 | Purchasing 1 tomato

4147 | Sched-A-0 | Done 1 tomato

4147 | Sched-A-0 | Purchasing 1 cheese

5148 | Sched-A-0 | Done 1 cheese

Notice how each product blocks subsequent ones from processing. When the purchase of bread is done, butter begins immediately, but not earlier. Strangely, even replacing map() with flatMap() does not help, and the output is exactly the same:

Observable

.just("bread", "butter", "milk", "tomato", "cheese")

.subscribeOn(schedulerA)

.flatMap(prod -> rxGroceries.purchase(prod, 1))

.reduce(BigDecimal::add)

.single();

The code does not work concurrently because there is just a single flow of events, which by design must run sequentially. Otherwise, your Subscriber would need to be aware of concurrent notifications (onNext(), onComplete(), etc.), so it is a fair compromise. Luckily, the idiomatic solution is very close. The main Observable emitting products cannot be parallelized. However, for each product, we create a new, independent Observable as returned from purchase(). Because they are independent, we can safely schedule each one of them concurrently:

Observable<BigDecimal> totalPrice = Observable

.just("bread", "butter", "milk", "tomato", "cheese")

.flatMap(prod ->

rxGroceries

.purchase(prod, 1)

.subscribeOn(schedulerA))

.reduce(BigDecimal::add)

.single();

Can you spot where subscribeOn() is? The main Observable is not really doing anything, so a special thread pool is unnecessary. However each substream created within flatMap() is supplied with a schedulerA. Every time subscribeOn() is used to the Scheduler gets a chance to return a new Worker, and therefore a separate thread (simplifying a bit):

113 | Sched-A-1 | Purchasing 1 butter

114 | Sched-A-0 | Purchasing 1 bread

125 | Sched-A-2 | Purchasing 1 milk

125 | Sched-A-3 | Purchasing 1 tomato

126 | Sched-A-4 | Purchasing 1 cheese

1126 | Sched-A-2 | Done 1 milk

1126 | Sched-A-0 | Done 1 bread

1126 | Sched-A-1 | Done 1 butter

1128 | Sched-A-3 | Done 1 tomato

1128 | Sched-A-4 | Done 1 cheese

Finally, we achieved true concurrency. Each purchase operation now begins at the same time and they all eventually finish. The flatMap() operator is carefully designed and implemented so that it collects all events from all independent streams and pushes them downstream sequentially. However, as we already learned in [“Order of Events After flatMap()”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch03.html#order-of-events-after-flatmap), we can no longer rely on the order of downstream events—they neither begin nor complete in the same order as they were emitted (the original sequence began at bread). When events reach the reduce() operator, they are already sequential and well behaving.

By now, you should slowly move away from the classic Thread model and understand how Schedulers work. But if you find it difficult, here is a simple analogy:

* Observable without any Scheduler works like a single-threaded program with blocking method calls passing data between one another.
* Observable with a single subscribeOn() is like starting a big task in the background Thread. The program within that Thread is still sequential, but at least it runs in the background.
* Observable using flatMap() where each internal Observable has subscribeOn() works like ForkJoinPool from java.util.concurrent, where each substream is a fork of execution and flatMap() is a safe join stage.

Of course, the preceding tips only apply to blocking Observables, which are rarely seen in real applications. If your underlying Observables are already asynchronous, achieving concurrency is a matter of understanding how they are combined and when subscription occurs. For example, merge() on two streams will subscribe to both of them concurrently, whereas the concat() operator waits until the first stream finishes before it subscribes to the second one.

## Batching Requests Using groupBy()

Did you notice that RxGroceries.purchase() takes productName and quantity even though the quantity was always one? What if our grocery list had some products multiple times, indicating bigger demand? The first naive implementation simply sends the same request—for example, for egg, multiple times, each time asking for one. Fortunately, we can declaratively batch such requests by using groupBy()—and this still works with declarative concurrency:

import org.apache.commons.lang3.tuple.Pair;

Observable<BigDecimal> totalPrice = Observable

.just("bread", "butter", "egg", "milk", "tomato",

"cheese", "tomato", "egg", "egg")

.groupBy(prod -> prod)

.flatMap(grouped -> grouped

.count()

.map(quantity -> {

String productName = grouped.getKey();

return Pair.of(productName, quantity);

}))

.flatMap(order -> store

.purchase(order.getKey(), order.getValue())

.subscribeOn(schedulerA))

.reduce(BigDecimal::add)

.single();

This code is quite complex, so before revealing the output, let’s quickly go through it. First, we group products simply by their name, thus identity function prod -> prod. In return we get an awkward Observable<GroupedObservable<String, String>>. There is nothing wrong with that. Next, flatMap() receives each GroupedObservable<String, String>, representing all products of the same name. So, for example, there will be an ["egg", "egg", "egg"] Observable there with a key "egg", as well. If groupBy() used a different key function, like prod.length(), the same sequence would have a key 3.

At this point, within flatMap() we need to construct an Observable of type Pair<String, Integer> which represents every unique product and its quantity. Both count() and map() return an Observable, so everything lines up perfectly. Second flatMap() receives order of type Pair<String, Integer> and makes a purchase, this time the quantity can be bigger. The output looks perfect; notice that bigger orders are slightly slower, but still it is much faster than having several repeated requests:

164 | Sched-A-0 | Purchasing 1 bread

165 | Sched-A-1 | Purchasing 1 butter

166 | Sched-A-2 | Purchasing 3 egg

166 | Sched-A-3 | Purchasing 1 milk

166 | Sched-A-4 | Purchasing 2 tomato

166 | Sched-A-5 | Purchasing 1 cheese

1151 | Sched-A-0 | Done 1 bread

1178 | Sched-A-1 | Done 1 butter

1180 | Sched-A-5 | Done 1 cheese

1183 | Sched-A-3 | Done 1 milk

1253 | Sched-A-4 | Done 2 tomato

1354 | Sched-A-2 | Done 3 egg

If you believe that your system can benefit from batching this way or the other, check out [“Batching and Collapsing Commands”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch08.html#Batching-and-collapsing-commands).

## Declarative Concurrency with observeOn()

Believe it or not, concurrency in RxJava can be described by two operators: the aformentioned subscribeOn() and observeOn(). They seem very similar and are confusing to newcomers, but their semantics are actually quite clear and reasonable.

subscribeOn() allows choosing which Scheduler will be used to invoke OnSubscribe (lambda expression inside create()). Therefore, any code inside create() is pushed to a different thread—for example, to avoid blocking the main thread. Conversely, observeOn() controls which Scheduler is used to invoke downstream Subscribers occurring after observeOn(). For example, calling create() happens in the io() Scheduler (via subscribeOn(io())) to avoid blocking the user interface. However, updating the user interface widgets must happen in the UI thread (both Swing and Android have this constraint), so we use observeOn() for example with AndroidSchedulers.mainThread() before operators or subscribers changing UI. This way we can use one Scheduler to handle create() and all operators up to the first observeOn(), but other(s) to apply transformations. This is best explained with an example:

log("Starting");

final Observable<String> obs = simple();

log("Created");

obs

.doOnNext(x -> log("Found 1: " + x))

.observeOn(schedulerA)

.doOnNext(x -> log("Found 2: " + x))

.subscribe(

x -> log("Got 1: " + x),

Throwable::printStackTrace,

() -> log("Completed")

);

log("Exiting");

observeOn() occurs somewhere in the pipeline chain, and this time, as opposed to subscribeOn(), the position of observeOn() is quite important. No matter what Scheduler was running operators above observeOn() (if any), everything below uses the supplied Scheduler. In this example, there is no subscribeOn(), so the default is applied (no concurrency):

23 | main | Starting

136 | main | Created

163 | main | Subscribed

163 | main | Found 1: A

163 | main | Found 1: B

163 | main | Exiting

163 | Sched-A-0 | Found 2: A

164 | Sched-A-0 | Got 1: A

164 | Sched-A-0 | Found 2: B

164 | Sched-A-0 | Got 1: B

164 | Sched-A-0 | Completed

All of the operators above observeOn are executed within client thread, which happens to be the default in RxJava. But below observeOn(), the operators are executed within the supplied Scheduler. This will become even more obvious when both subscribeOn() and multiple observeOn() occur within the pipeline:

log("Starting");

final Observable<String> obs = simple();

log("Created");

obs

.doOnNext(x -> log("Found 1: " + x))

.observeOn(schedulerB)

.doOnNext(x -> log("Found 2: " + x))

.observeOn(schedulerC)

.doOnNext(x -> log("Found 3: " + x))

.subscribeOn(schedulerA)

.subscribe(

x -> log("Got 1: " + x),

Throwable::printStackTrace,

() -> log("Completed")

);

log("Exiting");

Can you predict the output? Remember, everything below observeOn() is run within the supplied Scheduler, of course until another observeOn() is encountered. Additionally subscribeOn() can occur anywhere between Observable and subscribe(), but this time it only affects operators down to the first observeOn():

21 | main | Starting

98 | main | Created

108 | main | Exiting

129 | Sched-A-0 | Subscribed

129 | Sched-A-0 | Found 1: A

129 | Sched-A-0 | Found 1: B

130 | Sched-B-0 | Found 2: A

130 | Sched-B-0 | Found 2: B

130 | Sched-C-0 | Found 3: A

130 | Sched-C-0 | Got: A

130 | Sched-C-0 | Found 3: B

130 | Sched-C-0 | Got: B

130 | Sched-C-0 | Completed

Subscription occurs in schedulerA because that is what we specified in subscribeOn(). Also "Found 1" operator was executed within that Scheduler because it is before the first observeOn(). Later, the situation becomes more interesting. observeOn() switches current Scheduler to schedulerB, and "Found 2" is using this one, instead. The last observeOn(schedulerC) affects both "Found 3" operator as well as Subscriber. Remember that Subscriber works within the context of the last encountered Scheduler.

subscribeOn() and observeOn() work really well together when you want to physically decouple producer (Observable.create()) and consumer (Subscriber). By default, there is no such decoupling, and RxJava simply uses the same thread. subscribeOn() only is not enough, we simply choose a different thread. observeOn() is better, but then we block the client thread in case of synchronous Observables. Because most of the operators are nonblocking and lambda expressions used inside them tend to be short and cheap, typically there is just one subscribeOn() and observeOn() in the pipeline of operators. subscribeOn() can be placed close to the original Observable to improve readability, whereas observeOn() is close to subscribe() so that only Subscriber uses that special Scheduler, other operators rely on the Scheduler from subscribeOn().

Here is a more advanced program that takes advantage of these two operators:

log("Starting");

Observable<String> obs = Observable.create(subscriber -> {

log("Subscribed");

subscriber.onNext("A");

subscriber.onNext("B");

subscriber.onNext("C");

subscriber.onNext("D");

subscriber.onCompleted();

});

log("Created");

obs

.subscribeOn(schedulerA)

.flatMap(record -> store(record).subscribeOn(schedulerB))

.observeOn(schedulerC)

.subscribe(

x -> log("Got: " + x),

Throwable::printStackTrace,

() -> log("Completed")

);

log("Exiting");

Where store() is a simple nested operation:

Observable<UUID> store(String s) {

return Observable.create(subscriber -> {

log("Storing " + s);

//hard work

subscriber.onNext(UUID.randomUUID());

subscriber.onCompleted();

});

}

The production of events occurs in schedulerA, but each event is processed independently using schedulerB to improve concurrency, a technique we learned in [“subscribeOn() Concurrency and Behavior”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#subscribeOn-concurrency-and-behavior). The subscription in the end happens in yet another schedulerC. We are pretty sure you understand by now which Scheduler/thread will execute which action, but just in case (empty lines added for clarity):

26 | main | Starting

93 | main | Created

121 | main | Exiting

122 | Sched-A-0 | Subscribed

124 | Sched-B-0 | Storing A

124 | Sched-B-1 | Storing B

124 | Sched-B-2 | Storing C

124 | Sched-B-3 | Storing D

1136 | Sched-C-1 | Got: 44b8b999-e687-485f-b17a-a11f6a4bb9ce

1136 | Sched-C-1 | Got: 532ed720-eb35-4764-844e-690327ac4fe8

1136 | Sched-C-1 | Got: 13ddf253-c720-48fa-b248-4737579a2c2a

1136 | Sched-C-1 | Got: 0eced01d-3fa7-45ec-96fb-572ff1e33587

1137 | Sched-C-1 | Completed

observeOn() is especially important for applications with a UI for which we do not want to block the UI event-dispatching thread. On Android (see [“Android Development with RxJava”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch08.html#android)) or Swing, some actions like updating the UI must be executed within a specific thread. But doing too much in that thread renders your UI unresponsive. In these cases, you put observeOn() close to subscribe() so that code within the subscription is invoked within the context of a particular Scheduler (like UI-thread). However, other transformations, even rather cheap, should be executed outside UI thread. On the server, observeOn() is seldom used because the true source of concurrency is built into most Observables. This leads to an interesting conclusion: RxJava controls concurrency with just two operators (subscribeOn() and observeOn()), but the more you use reactive extensions, the less frequently you will see these in production code.

## Other Uses for Schedulers

There are numerous operators that by default use some Scheduler. Typically, Schedulers.computation() is used if none is supplied—JavaDoc always makes it clear. For example, the delay() operator takes upstream events and pushes them downstream after a given time. Obviously, it cannot hold the original thread during that period, so it must use a different Scheduler:

Observable

.just('A', 'B')

.delay(1, SECONDS, schedulerA)

.subscribe(this::log);

Without supplying a custom schedulerA, all operators below delay() would use the computation() Scheduler. There is nothing inherently wrong with that; however, if your Subscriber is blocked on I/O it would consume one Worker from globally shared computation() scheduler, possibly affecting the entire system. Other important operators that support custom Scheduler are: interval(), range(), timer(), repeat(), skip(), take(), timeout(), and several others that have yet to be introduced. If you do not provide a scheduler to such operators, computation() Scheduler is utilized, which is a safe default in most cases.

Mastering schedulers is essential to writing scalable and safe code using RxJava. The difference between subscribeOn() and observeOn() is especially important under high load where every task must be executed precisely when we expect. In truly reactive applications, for which all long-running operations are asynchronous, very few threads and thus Schedulers are needed. But there is always this one API or dependency that requires blocking code.

Last but not least, we must be sure that Schedulers used downstream can keep up with the load generated by Schedulers upstream. But this danger will be explained in great detail in [Chapter 6](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch06.html#Flow-control-and-backpressure).

# Summary

This chapter described several patterns in traditional applications that can be replaced with RxJava. I hope you understand by now that high-frequency trading or streaming posts from social media are not the only use cases for RxJava. As a matter of fact, almost any API can be seamlessly replaced with Observable. Even if you don’t want or need the power of reactive extensions at the moment, it will allow you to evolve implementation without introducing backward-incompatible changes. Moreover, it is the client that eventually harvests all the possibilities offered by RxJava, like laziness, declarative concurrency, or asynchronous chaining. Even better, because of seamless conversion from Observable to BlockingObservable, traditional clients can consume your API as they want, and you can always provide a simple bridge layer.

You should be fairly confident with RxJava and understand the benefits of applying it even in legacy systems. Undoubtedly, working with reactive Observables is more challenging and has a somewhat steep learning curve. But the advantages and possibilities of growth simply can’t be exaggerated. Imagine if we could write entire applications using reactive extensions, from top to bottom? Like a greenfield project for which we have control over every API, interface, and external system. [Chapter 5](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch05.html#Going-fully-reactive) will discuss how you can write such an application and what the implications are.

[1](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#idm140239472529008-marker) In fact, RxJava tries to stay on the same thread via thread affinity in the event loop model to take advantage of this, as well.

[2](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#idm140239472528016-marker) See also [“Bulkhead Pattern and Fail-Fast”](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch08.html#Bulkhead-pattern-and-Fail-fast)

[3](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#idm140239471172304-marker) Compare it to lazy evaluation of expressions in Haskell.

[4](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#idm140239470771152-marker) Obviously, for any real project, you will use a production-grade logging system like [Logback](http://logback.qos.ch/) or [Log4J 2](http://logging.apache.org/log4j/2.x/).

[5](https://www.safaribooksonline.com/library/view/reactive-programming-with/9781491931646/ch04.html#idm140239470096960-marker) <https://github.com/ReactiveX/RxSwing>

# [Notes on Reactive Programming Part II: Writing Some Code](https://spring.io/blog/2016/06/13/notes-on-reactive-programming-part-ii-writing-some-code)

In this article we continue the series on [Reactive Programming](https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape), and we concentrate on explaining some concepts through actual code samples. The end result should be that you understand a bit better what makes Reactive different, and what makes it functional. The examples here are quite abstract, but they give you a way to think about the APIs and the programming style, and start to get a feel for how it is different. We will see the elements of Reactive, and learn how to control the flow of data, and process in background threads if necessary.

## Setting Up a Project

We will use the Reactor libraries to illustrate the points we need to make. The code could just as easily be written with other tools. If you want to play with the code and see it working without having to copy-paste anything, there are working samples with tests in [Github](https://github.com/dsyer/reactive-notes).

To get started grab a blank project from <https://start.spring.io> and add the Reactor Core dependency. With Maven

<dependency>

<groupId>io.projectreactor</groupId>

<artifactId>reactor-core</artifactId>

<version>3.0.0.RC2</version>

</dependency>

With Gradle it’s very similar:

compile 'io.projectreactor:reactor-core:3.0.0.RC2'

Now let’s write some code.

## What Makes it Functional?

The basic building block of Reactive is a sequence of events, and two protagonists, a publisher and a subscriber to those events. It’s also OK to call a sequence a "stream" because that’s what it is. If we need to, we will use the word "stream" with a small "s", but Java 8 has a java.util.Stream which is different, so try not to get confused. We will try to concentrate the narrative on the publisher and subscriber anyway (that’s what Reactive Streams does).

Reactor is the library we are going to use in samples, so we’ll stick to the notation there, and call the publisher a Flux (it implements the interface Publisher from Reactive Streams). The RxJava library is very similar and has a lot of parallel features, so in that case we would be talking about an Observable instead, but the code would be very similar. (Reactor 2.0 called it a Stream which is confusing if we need to talk about Java 8 Streams as well, so we’ll only use the new code in Reactor 3.0.)

### Generators

A Flux is a publisher of a sequence of events of a specific POJO type, so it is generic, i.e. Flux<T> is a publisher of T. Flux has some static convenience methods to create instances of itself from a variety of sources. For example, to create a Flux from an array:

Flux<String> flux = Flux.just("red", "white", "blue");

We just generated a Flux, and now we can do stuff with it. There are actually only two things you can do with it: operate on it (transform it, or combine it with other sequences), subscribe to it (it’s a publisher).

### Single Valued Sequences

Often you encounter a sequence that you know has only one or zero elements, for example a repository method that finds an entity by its id. Reactor has a Mono type representing a single valued or empty Flux. Mono has a very similar API to Flux but more focused because not all operators make sense for single-valued sequences. RxJava also has a bolt on (in version 1.x) called Single, and also Completable for an empty sequence. The empty sequence in Reactor is Mono<Void>.

### Operators

There are a lot of methods on a Flux and nearly all of them are operators. We aren’t going to look at them all here because there are better places to look for that (like the Javadocs). We only need to get a flavour for what an operator is, and what it can do for you.

For instance, to ask for the internal events inside a Flux to be logged to standard out, you can call the .log() method. Or you can transform it using a map():

Flux<String> flux = Flux.just("red", "white", "blue");

Flux<String> upper = flux

.log()

.map(String::toUpperCase);

In this code we transformed the strings in the input by converting them to upper case. So far, so trivial.

What’s interesting about this little sample — mind blowing, even, if you’re not used to it — is that no data have been processed yet. Nothing has even been logged because literally, nothing happened (try it and you will see). Calling operators on a Flux amounts to building a plan of execution for later. It is completely declarative, and it’s why people call it "functional". The logic implemented in the operators is only executed when data starts to flow, and that doesn’t happen until someone subscribes to the Flux (or equivalently to the Publisher).

The same declarative, functional approach to processing a sequence of data exists in all Reactive libraries, and also in Java 8 Streams. Consider this, similar looking code, using a Stream with the same contents as the Flux:

Stream<String> stream = Streams.of("red", "white", "blue");

Stream<String> upper = stream.map(value -> {

System.out.println(value);

return value.toUpperCase();

});

The observation we made about Flux applies here: no data is processed, it’s just a plan of execution. There are, however, some important differences between Flux and Stream, which make Stream an inappropriate API for Reactive use cases. Flux has a lot more operators, much of which is just convenience, but the real difference comes when you want to consume the data, so that’s what we need to look at next.

|  |  |
| --- | --- |
| Tip | There is a useful blog by Sebastien Deleuze on [Reactive Types](https://spring.io/blog/2016/04/19/understanding-reactive-types), where he describes the differences between the various streaming and reactive APIs by looking at the types they define, and how you would use them. The differences between Flux and Stream are highlighted there in more detail. |

### Subscribers

To make the data flow you have to subscribe to the Flux using one of the subscribe() methods. Only those methods make the data flow. They reach back through the chain of operators you declared on your sequence (if any) and request the publisher to start creating data. In the sample samples we have been working with, this means the underlying collection of strings is iterated. In more complicated use case it might trigger a file to be read from the filesystem, or a pull from a database or a call to an HTTP service.

Here’s a call to subscribe() in action:

Flux.just("red", "white", "blue")

.log()

.map(String::toUpperCase)

.subscribe();

The output is:

09:17:59.665 [main] INFO reactor.core.publisher.FluxLog - onSubscribe(reactor.core.publisher.FluxIterable$IterableSubscription@3ffc5af1)

09:17:59.666 [main] INFO reactor.core.publisher.FluxLog - request(unbounded)

09:17:59.666 [main] INFO reactor.core.publisher.FluxLog - onNext(red)

09:17:59.667 [main] INFO reactor.core.publisher.FluxLog - onNext(white)

09:17:59.667 [main] INFO reactor.core.publisher.FluxLog - onNext(blue)

09:17:59.667 [main] INFO reactor.core.publisher.FluxLog - onComplete()

So we can see from this that the effect of subscribe() without an argument, is to request the publisher to send all data — there’s only one request() logged and it’s "unbounded". We can also see callbacks for each item that is published (onNext()), for the end of the sequence (onComplete()), and for the original subscription (onSubscribe()). If you needed to you could listen for those events yourself using the doOn\*() methods in Flux, which are themselves operators, not subscribers, so they don’t cause any data to flow on their own.

The subscribe() method is overloaded, and the other variants give you different options to control what happens. One important and convenient form is subscribe() with callbacks as arguments. The first argument is a Consumer, which gives you a callback with each of the items, and you can also optionally add a Consumer for an error if there is one, and a vanilla Runnable to execute when the sequence is complete. For example, just with the per-item callback:

Flux.just("red", "white", "blue")

.log()

.map(String::toUpperCase)

.subscribe(System.out::println);

Here’s the output:

09:56:12.680 [main] INFO reactor.core.publisher.FluxLog - onSubscribe(reactor.core.publisher.FluxArray$ArraySubscription@59f99ea)

09:56:12.682 [main] INFO reactor.core.publisher.FluxLog - request(unbounded)

09:56:12.682 [main] INFO reactor.core.publisher.FluxLog - onNext(red)

RED

09:56:12.682 [main] INFO reactor.core.publisher.FluxLog - onNext(white)

WHITE

09:56:12.682 [main] INFO reactor.core.publisher.FluxLog - onNext(blue)

BLUE

09:56:12.682 [main] INFO reactor.core.publisher.FluxLog - onComplete()

We could control the flow of data, and make it "bounded", in a variety of ways. The raw API for control is the Subscription you get from a Subscriber. The equivalent long form of the short call to subscribe() above is:

.subscribe(new Subscriber<String>() {

@Override

public void onSubscribe(Subscription s) {

s.request(Long.MAX\_VALUE);

}

@Override

public void onNext(String t) {

System.out.println(t);

}

@Override

public void onError(Throwable t) {

}

@Override

public void onComplete() {

}

});

To control the flow, e.g. to consume at most 2 items at a time, you could use the Subscription more intelligently:

.subscribe(new Subscriber<String>() {

private long count = 0;

private Subscription subscription;

@Override

public void onSubscribe(Subscription subscription) {

this.subscription = subscription;

subscription.request(2);

}

@Override

public void onNext(String t) {

count++;

if (count>=2) {

count = 0;

subscription.request(2);

}

}

...

This Subscriber is "batching" items 2 at a time. It’s a common use case so you might think about extracting the implementation to a convenience class, and that would make the code more readable too. The output looks like this:

09:47:13.562 [main] INFO reactor.core.publisher.FluxLog - onSubscribe(reactor.core.publisher.FluxArray$ArraySubscription@61832929)

09:47:13.564 [main] INFO reactor.core.publisher.FluxLog - request(2)

09:47:13.564 [main] INFO reactor.core.publisher.FluxLog - onNext(red)

09:47:13.565 [main] INFO reactor.core.publisher.FluxLog - onNext(white)

09:47:13.565 [main] INFO reactor.core.publisher.FluxLog - request(2)

09:47:13.565 [main] INFO reactor.core.publisher.FluxLog - onNext(blue)

09:47:13.565 [main] INFO reactor.core.publisher.FluxLog - onComplete()

In fact the batching subscriber is such a common use case that there are convenience methods already available in Flux. The batching example above can be implemented like this:

Flux.just("red", "white", "blue")

.log()

.map(String::toUpperCase)

.subscribe(null, 2);

(note the call to subscribe() with a request limit). Here’s the output:

10:25:43.739 [main] INFO reactor.core.publisher.FluxLog - onSubscribe(reactor.core.publisher.FluxArray$ArraySubscription@4667ae56)

10:25:43.740 [main] INFO reactor.core.publisher.FluxLog - request(2)

10:25:43.740 [main] INFO reactor.core.publisher.FluxLog - onNext(red)

10:25:43.741 [main] INFO reactor.core.publisher.FluxLog - onNext(white)

10:25:43.741 [main] INFO reactor.core.publisher.FluxLog - request(2)

10:25:43.741 [main] INFO reactor.core.publisher.FluxLog - onNext(blue)

10:25:43.741 [main] INFO reactor.core.publisher.FluxLog - onComplete()

|  |  |
| --- | --- |
| Tip | A library that will process sequences for you, like Spring Reactive Web, can handle the subscriptions. It’s good to be able to push these concerns down the stack because it saves you from cluttering your code with non-business logic, making it more readable and easier to test and maintain. So as a rule, it is a good thing if you can **avoid subscribing** to a sequence, or at least push that code into a processing layer, and out of the business logic. |

### Threads, Schedulers and Background Processing

An interesting feature of all the logs above is that they are all on the "main" thread, which is the thread of the caller to subscribe(). This highlights an important point: Reactor is extremely frugal with threads, because that gives you the greatest chance of the best possible performance. That might be a surprising statement if you’ve been wrangling threads and thread pools and asynchronous executions for the last 5 years, trying to squeeze more juice out of your services. But it’s true: in the absence of any imperative to switch threads, even if the JVM is optimized to handle threads very efficiently, it is always faster to do computation on a single thread. Reactor has handed you the keys to control all the asynchronous processing, and it assumes you know what you are doing.

Flux provides a few configurer methods that control the thread boundaries. For example, you can configure the subscriptions to be handled in a background thread using Flux.subscribeOn():

Flux.just("red", "white", "blue")

.log()

.map(String::toUpperCase)

.subscribeOn(Schedulers.parallel())

.subscribe(null, 2);

the result can be seen in the output:

13:43:41.279 [parallel-1-1] INFO reactor.core.publisher.FluxLog - onSubscribe(reactor.core.publisher.FluxArray$ArraySubscription@58663fc3)

13:43:41.280 [parallel-1-1] INFO reactor.core.publisher.FluxLog - request(2)

13:43:41.281 [parallel-1-1] INFO reactor.core.publisher.FluxLog - onNext(red)

13:43:41.281 [parallel-1-1] INFO reactor.core.publisher.FluxLog - onNext(white)

13:43:41.281 [parallel-1-1] INFO reactor.core.publisher.FluxLog - request(2)

13:43:41.281 [parallel-1-1] INFO reactor.core.publisher.FluxLog - onNext(blue)

13:43:41.281 [parallel-1-1] INFO reactor.core.publisher.FluxLog - onComplete()

|  |  |
| --- | --- |
| Tip | if you write this code yourself, or copy-paste it, remember to wait for the processing to stop before the JVM exits. |

Note that the subscription, and all the processing, takes place on a single background thread "parallel-1-1" — this is because we asked for the subscriber to our main Flux to be in the background. This is fine if the item processing is CPU intensive (but pointless being in a background thread, in point of fact, since you pay for the context switch but don’t get the results any faster). You might also want to be able to perform item processing that is I/O intensive and possibly blocking. In this case, you would want to get it done as quickly as possible without blocking the caller. A thread pool is still your friend, and that’s what you get from Schedulers.parallel(). To switch the processing of the individual items to separate threads (up to the limit of the pool) we need to break them out into separate publishers, and for each of those publishers ask for the result in a background thread. One way to do this is with an operator called flatMap(), which maps the items to a Publisher (potentially of a different type), and then back to a sequence of the new type:

Flux.just("red", "white", "blue")

.log()

.flatMap(value ->

Mono.just(value.toUpperCase())

.subscribeOn(Schedulers.parallel()),

2)

.subscribe(value -> {

log.info("Consumed: " + value);

})

Note here the use of flatMap() to push the items down into a "child" publisher, where we can control the subscription per item instead of for the whole sequence. Reactor has built in default behaviour to hang onto a single thread as long as possible, so we need to be explicit if we want it to process specific items or groups of items in a background thread. Actually, this is one of a handful of recognized tricks for forcing parallel processing (see the [Reactive Gems](https://github.com/reactor/reactive-streams-commons/issues/21) issue for more detail).

The output looks like this:

15:24:36.596 [main] INFO reactor.core.publisher.FluxLog - onSubscribe(reactor.core.publisher.FluxIterable$IterableSubscription@6f1fba17)

15:24:36.610 [main] INFO reactor.core.publisher.FluxLog - request(2)

15:24:36.610 [main] INFO reactor.core.publisher.FluxLog - onNext(red)

15:24:36.613 [main] INFO reactor.core.publisher.FluxLog - onNext(white)

15:24:36.613 [parallel-1-1] INFO com.example.FluxFeaturesTests - Consumed: RED

15:24:36.613 [parallel-1-1] INFO reactor.core.publisher.FluxLog - request(1)

15:24:36.613 [parallel-1-1] INFO reactor.core.publisher.FluxLog - onNext(blue)

15:24:36.613 [parallel-1-1] INFO reactor.core.publisher.FluxLog - onComplete()

15:24:36.614 [parallel-3-1] INFO com.example.FluxFeaturesTests - Consumed: BLUE

15:24:36.617 [parallel-2-1] INFO com.example.FluxFeaturesTests - Consumed: WHITE

Notice that there are now multiple threads consuming the items, and the concurrency hint in the flatMap() ensures that there are 2 items being processed at any given time, as long as they are available. We see request(1) a lot because the system is trying to keep 2 items in the pipeline, and generally they don’t finish processing at the same time. Reactor tries to be very smart here in fact, and it pre-fetches items from the upstream Publisher to try to eliminate waiting time for the subscriber (we aren’t seeing that here because the numbers are low — we are only processing 3 items).

|  |  |
| --- | --- |
| Tip | Three items ("red", "white", "blue") might be too few to convincingly see more than one background thread, so it might be better to generate more data. You could do that with a random number generator, for instance. |

Flux also has a publishOn() method which is the same, but for the listeners (i.e. onNext() or consumer callbacks) instead of for the subscriber itself:

Flux.just("red", "white", "blue")

.log()

.map(String::toUpperCase)

.subscribeOn(Schedulers.newParallel("sub"))

.publishOn(Schedulers.newParallel("pub"), 2)

.subscribe(value -> {

log.info("Consumed: " + value);

});

The output looks like this:

15:12:09.750 [sub-1-1] INFO reactor.core.publisher.FluxLog - onSubscribe(reactor.core.publisher.FluxIterable$IterableSubscription@172ed57)

15:12:09.758 [sub-1-1] INFO reactor.core.publisher.FluxLog - request(2)

15:12:09.759 [sub-1-1] INFO reactor.core.publisher.FluxLog - onNext(red)

15:12:09.759 [sub-1-1] INFO reactor.core.publisher.FluxLog - onNext(white)

15:12:09.770 [pub-1-1] INFO com.example.FluxFeaturesTests - Consumed: RED

15:12:09.771 [pub-1-1] INFO com.example.FluxFeaturesTests - Consumed: WHITE

15:12:09.777 [sub-1-1] INFO reactor.core.publisher.FluxLog - request(2)

15:12:09.777 [sub-1-1] INFO reactor.core.publisher.FluxLog - onNext(blue)

15:12:09.777 [sub-1-1] INFO reactor.core.publisher.FluxLog - onComplete()

15:12:09.783 [pub-1-1] INFO com.example.FluxFeaturesTests - Consumed: BLUE

Notice that the consumer callbacks (logging "Consumed: …​") are on the publisher thread pub-1-1. If you take out the subscribeOn() call, you might see all of the 2nd chunk of data processed on the pub-1-1 thread as well. This, again, is Reactor being frugal with threads — if there’s no explicit request to switch threads it stays on the same one for the next call, whatever that is.

|  |  |
| --- | --- |
| Note | We changed the code in this sample from subscribe(null, 2) to adding a prefetch=2 to the publishOn(). In this case the fetch size hint in subscribe() would have been ignored. |

### Extractors: The Subscribers from the Dark Side

There is another way to subscribe to a sequence, which is to call Mono.block() or Mono.toFuture() or Flux.toStream() (these are the "extractor" methods — they get you out of the Reactive types into a less flexible, blocking abstraction). Flux also has converters collectList() and collectMap() that convert from Flux to Mono. They don’t actually subscribe to the sequence, but they do throw away any control you might have had over the suscription at the level of the individual items.

|  |  |
| --- | --- |
| Warning | A good rule of thumb is "**never call an extractor**". There are some exceptions (otherwise the methods would not exist). One notable exception is in tests because it’s useful to be able to block to allow results to accumulate. |

These methods are there as an escape hatch to bridge from Reactive to blocking; if you need to adapt to a legacy API, for instance Spring MVC. When you call Mono.block() you throw away all the benefits of the Reactive Streams. This is the key difference between Reactive Streams and Java 8 Streams — the native Java Stream only has the "all or nothing" subscription model, the equivalent of Mono.block(). Of course subscribe() can block the calling thread as well, so it’s just as dangerous as the converter methods, but you have more control — you can prevent it from blocking by using subscribeOn() and you can drip the items through by applying back pressure and periodically deciding whether to continue.

## Conclusion

In this article we have covered the basics of the Reactive Streams and Reactor APIs. If you need to know more there are plenty of places to look, but there’s no substitute for hands on coding, so use the code in [GitHub](https://github.com/dsyer/reactive-notes) (for this article in tests in the project called "flux"), or head over to the [Lite RX Hands On](https://github.com/reactor/lite-rx-api-hands-on) workshop. So far, really this is just overhead, and we haven’t learned much that we couldn’t have done in a more obvious way using non-Reactive tools. The [next article](https://spring.io/blog/2016/07/20/notes-on-reactive-programming-part-iii-a-simple-http-server-application) in the series will dig a little deeper into the blocking, dispatching and asynchronous sides of the Reactive model, and show you what opportunities there are to reap the real benefits.

# [Notes on Reactive Programming Part III: A Simple HTTP Server Application](https://spring.io/blog/2016/07/20/notes-on-reactive-programming-part-iii-a-simple-http-server-application)

In this article we continue the series on [Reactive Programming](https://spring.io/blog/2016/06/13/notes-on-reactive-programming-part-ii-writing-some-code), and the focus is less on learning the basic APIs and more on more concrete use cases and writing code that actually does something useful. We will see how Reactive is a useful abstraction for concurrent programming, but also that it has some very low level features that we should learn to treat with respect and caution. If we start to use these features to their full potential we can take control of layers in our application that previously were invisible, hidden by containers, platforms and frameworks.

## Bridging from Blocking to Reactive with Spring MVC

Being Reactive forces you to look at the world differently. Instead of asking for something and getting it (or not getting it), everything is delivered as a sequence (Publisher) and you have to subscribe to it. Instead of waiting for an answer, you have to register a callback. It’s not so hard when you get used to it, but unless the whole world turns on its head and becomes Reactive, you are going to find you need to interact with an old-style blocking API

Suppose we have a blocking method that returns an HttpStatus:

private RestTemplate restTemplate = new RestTemplate();

private HttpStatus block(int value) {

return this.restTemplate.getForEntity("http://example.com/{value}", String.class, value)

.getStatusCode();

}

and we want to call it repeatedly with different arguments and aggregate the results. It’s a classic "scatter-gather" use case, which you would get, for instance, if you had a paginated back end needed to summarize the "top N" items across multiple pages. Since the details of the non-reactive (blocking) operation are not relevant to the scatter-gather pattern, we can push them down into a method called block(), and implement it later. Here’s a (bad) example that calls the back end and aggregates into an object of type Result:

Flux.range(1, 10) **(1)**

.log()

.map(this::block) **(2)**

.collect(Result::new, Result::add) **(3)**

.doOnSuccess(Result::stop) **(4)**

1. make 10 calls
2. blocking code here
3. collect results and aggregate into a single object
4. at the end stop the clock (the result is a Mono<Result>)

Don’t do this at home. It’s a "bad" example because, while the APIs are technically being used correctly, we know that it is going to block the calling thread; this code is more or less equivalent to a for loop with the call to block() in the body of the loop. A better implementation would push the call to block() onto a background thread. We can do that by wrapping it in a method that returns a Mono<HttpStatus>:

private Mono<HttpStatus> fetch(int value) {

return Mono.fromCallable(() -> block(value)) **(1)**

.subscribeOn(this.scheduler); **(2)**

}

1. blocking code here inside a Callable to defer execution
2. subscribe to the slow publisher on a background thread

The scheduler is declared separately as a shared field: Scheduler scheduler = Schedulers.parallel(). Then we can declare that we want to flatMap() the sequence instead of using map():

Flux.range(1, 10)

.log()

.flatMap( **(1)**

this::fetch, 4) **(2)**

.collect(Result::new, Result::add)

.doOnSuccess(Result::stop)

1. drop down to a new publisher to process in parallel
2. concurrency hint in flatMap

### Embedding in a Non-Reactive Server

If we wanted to run the scatter-gather code above in a non-reactive server like a servlet container, we could use Spring MVC, like this:

@RequestMapping("/parallel")

public CompletableFuture<Result> parallel() {

return Flux.range(1, 10)

...

.doOnSuccess(Result::stop)

.toFuture();

}

If you read the Javadocs for @RequestMapping you will find that a method can return a CompletableFuture "which the application uses to produce a return value in a separate thread of its own choosing". The separate thread in this case is provided by "scheduler", which is a thread pool, so the processing is happening on multiple threads, 4 at a time because of the way that flatMap() is called.

### No Such Thing as a Free Lunch

The scatter-gather with a background thread is a useful pattern but it isn’t perfect — it’s not blocking the caller, but it’s blocking something, so it’s just moving the problem around. There are some practical implications. We have an HTTP server with (probably) non-blocking IO handlers, passing work back to a thread pool, one HTTP request per thread — all of this is happening inside a servlet container (e.g. Tomcat). The request is processed asynchronously, so the worker thread in Tomcat isn’t blocked, and the thread pool that we created in our "scheduler" is processing on up to 4 concurrent threads. We are processing 10 back end requests (calls to block()) so there is a maximum, theoretical benefit of using the scheduler of 4 times lower latency. In other words, if processing all 10 requests one after the other in a single thread takes 1000ms, we might see a processing time of 250ms for a single incoming request at our HTTP service. We should emphasise the "might" though: it’s only going to go that fast if there is no contention for the processing threads (in both stages, the Tomcat workers, and the application scheduler). If you have a server with a large number of cores, very low concurrency, i.e. a small number of clients connecting to your application, and hardly any chance that two will make a request at the same time, then you will probably see close to the theoretical improvement. As soon as there are multiple clients trying to connect, they will all be competing for the same 4 threads, and the latency will drift up, and could even be worse than that experienced by a single client with no background processing. We can improve the latency for concurrent clients by creating the scheduler with a larger thread pool, e.g.

private Scheduler scheduler = Schedulers.newParallel("sub", 16);

(16 threads.) Now we are using more memory for the threads and their stacks, and we can expect to see lower latency for low concurrency, but not necessarily for high concurrency if our hardware has fewer than 16 cores. We also do not expect higher throughput under load: if there is contention for the threads, there is a high cost for managing those resources and that has to be reflected somwehere in a metric that matters. If you are interested in more detailed analysis of that kind of trade off, some detailed analyses of how performance metrics scale under load can be found in a blog series by [Rob Harrop](https://robharrop.github.io/).

|  |  |
| --- | --- |
| Tip | Tomcat allocates 100 threads for processing HTTP requests by default. That is excessive if we know all the processing is going to be on our scheduler thread pool. There is an impedance mismatch: the scheduler thread pool can be a bottleneck because it has fewer threads than the upstream Tomcat thread pool. This highlights the fact that performance tuning can be very hard, and, while you might have control of all the configuration, it’s a delicate balance. |

We can do better than a fixed thread pool if we use a scheduler that adjusts its capacity according to demand. Reactor has a convenience for that, so if you try the same code with Schedulers.elastic() (you can call it anywhere; there is only one instance), you will see that under load more threads are created.

## Reactive all the Way Down

The bridge from blocking to reactive is a useful pattern, and is easy to implement using available technology in Spring MVC (as shown above). The next stage in the Reactive journey is to break out of blocking in application threads completely, and to do that requires new APIs and new tools. Ultimately we have to be Reactive all the way down the stack, including servers and clients. This is the goal of Spring Reactive, which is a new framework, orthogonal to Spring MVC, but meeting the same needs, and using a similar programming model.

|  |  |
| --- | --- |
| Note | Spring Reactive started as a standalone project, but is folded into the Spring Framework in version 5.0 (first milestone June 2016). |

The first step to fully Reactive in our scatter-gather example would be to replace spring-boot-starter-web with spring-boot-starter-web-reactive on the classpath. In Maven:

<dependencies>

<dependency>

<groupId>org.springframework.boot.experimental</groupId>

<artifactId>spring-boot-starter-web-reactive</artifactId>

</dependency>

...

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.boot.experimental</groupId>

<artifactId>spring-boot-dependencies-web-reactive</artifactId>

<version>0.1.0.M1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

or in Gradle:

dependencies {

compile('org.springframework.boot.experimental:spring-boot-starter-web-reactive')

...

}

dependencyManagement {

imports {

mavenBom "org.springframework.boot.experimental:spring-boot-dependencies-web-reactive:0.1.0.M1"

}

}

(At the time of writing there are snapshots and milestones of this experimental library in repo.spring.io, or you could build and install locally from [GitHub](https://github.com/bclozel/spring-boot-reactive-web).)

Then in the controller, we can simply lose the bridge to CompletableFuture and return an object of type Mono:

@RequestMapping("/parallel")

public Mono<Result> parallel() {

return Flux.range(1, 10)

.log()

.flatMap(this::fetch, 4)

.collect(Result::new, Result::add)

.doOnSuccess(Result::stop);

}

Take this code and put it in a Spring Boot application and it will run in Tomcat, Jetty or Netty, depending on what it finds on the classpath. Tomcat is the default server in that starter, so you have to exclude it and provide a different one if you want to switch. All three have very similar characteristics in terms of startup time, memory usage and runtime resource usage.

We still have the blocking backend call in block(), so we still have to subscribe on a thread pool to avoid blocking the caller. We can change that if we have a non-blocking client, e.g. instead of using RestTemplate we use the new WebClient then we might do this instead to use a non-blocking client:

private WebClient client = new WebClient(new ReactorHttpClientRequestFactory());

private Mono<HttpStatus> fetch(int value) {

return this.client.perform(HttpRequestBuilders.get("http://example.com"))

.extract(WebResponseExtractors.response(String.class))

.map(response -> response.getStatusCode());

}

Note that the WebClient.perform() (or the WebResponseExtractor to be precise) has a Reactive return type, which we have transformed into a Mono<HttpStatus>, but we have not subscribed to it. We want the framework to do all the subscribing, so now we are Reactive all the way down.

|  |  |
| --- | --- |
| Warning | Methods in Spring Reactive that return a Publisher **are** non-blocking, but in general a method that returns a Publisher (or Flux, Mono or Observable) is only a hint that it might be non-blocking. If you are writing such methods it is important to analyse (and preferably test) whether they block, and to let callers know explicitly if they might do. |
| Note | The trick we played just now of using a non-blocking client to simplify the HTTP stack works in regular Spring MVC as well. The result of the fetch() method above can be converted to a CompletableFuture and passed out of a regular @RequestMapping method (in Spring Boot 1.3 for instance). | |

### Inversion of Control

Now we can remove the concurrency hint after the call to fetch() in the HTTP request handler:

@RequestMapping("/netty")

public Mono<Result> netty() {

return Flux.range(1, 10) **(1)**

.log() //

.flatMap(this::fetch) **(2)**

.collect(Result::new, Result::add)

.doOnSuccess(Result::stop);

}

1. make 10 calls
2. drop down to a new publisher to process in parallel

Taking into account that we don’t need the extra callable and subscriber thread at all, this code is a lot cleaner than when we had to bridge to the blocking client, which can be attributed to the fact that the code is Reactive all the way down. The Reactive WebClient returns a Mono, and that drives us immediately to select flatMap() in the transformation chain, and the code we need just falls out. It’s a nicer experience to write it, and it’s more readable, so it’s easier to maintain. Also, since there is no thread pooling and no concurrency hint, there is no magic factor of 4 to plug into our performance expectations. There is a limit somewhere, but it’s not imposed by our choices in the application tier any more, nor is it limited by anything in the server "container". It’s not magic, and there are still laws of physics, so the backend calls are all still going to take 100ms or so each, but with low contention we might even see all 10 requests complete in roughly the same time it takes for one. As the load on the server increases latency and throughput will naturally degrade, but in a way that is governed by buffer contention and kernel networking, not by application thread management. It’s an inversion of control, to lower levels of the stack below the application code.

Remember the same application code runs on Tomcat, Jetty or Netty. Currently, the Tomcat and Jetty support is provided on top of Servlet 3.1 asynchronous processing, so it is limited to one request per thread. When the same code runs on the Netty server platform that constraint is lifted, and the server can dispatch requests sympathetically to the web client. As long as the client doesn’t block, everyone is happy. Performance metrics for the netty server and client probably show similar characteristics, but the Netty server is not restricted to processing a single request per thread, so it doesn’t use a large thread pool and we might expect to see some differences in resource utilization. We will come back to that later in another article in this series.

|  |  |  |
| --- | --- | --- |
| Tip | in the [sample code](https://github.com/dsyer/reactive-notes) the "reactive" sample has Maven profiles "tomcat", "tomcatNext" (for Tomcat 8.5), "jetty" and "netty", so you can easily try out all the different server options without changing a line of code. | |
| Note | the blocking code in many applications is not HTTP backend calls, but database interactions. Very few databases support non-blocking clients at this point in time (MongoDB and Couchbase are notable exceptions, but even those are not as mature as the HTTP clients). Thread pools and the blocking-to-reactive pattern will have a long life until all the database vendors catch up on the client side. |

### Still No Free Lunch

We have whittled down our basic scatter-gather use case until the code is very clean, and very sympathetic to the hardware it runs on. We wrote some simple code and it was stacked up and orchestrated very nicely into a working HTTP service using Spring. On a sunny day everyone is more than happy with the outcome. But as soon as there are errors, e.g. a badly behaved network connection, or a back end service that suffers from poor latency, we are going to suffer.

The first, most obvious way to suffer is that the code we wrote is declarative, so it’s hard to debug. When errors occur the diagnostics can be very opaque. Using the raw, low-level APIs, like Reactor without Spring, or even down to the level of Netty without Reactor would probably make it even worse, because then we would have to build a lot of error handling ourselves, repeating the boiler plate every time we interact with the network. At least with Spring and Reactor in the mix we can expect to see stack traces logged for stray, uncaught exceptions. They might not be easy to understand though because they happen on threads that we don’t control, and they sometimes show up as quite low level concerns, from unfamiliar parts of the stack.

Another source of pain is that if we ever make a mistake and block in one of our Reactive callbacks, we will be holding up **all** requests on the same thread. With the servlet-based containers every request is isolated to a thread, and blocking doesn’t hold up other requests because they are be processed on different threads. Blocking all requests is still a recipe for trouble, but it only shows up as increased latency with roughly a constant factor per request. In the Reactive world, blocking a single request can lead to increased latency for all requests, and blocking all requests can bring a server to its knees because the extra layers of buffers and threads are not there to take up the slack.

## Conclusion

It’s nice to be able to control all the moving parts in our asynchronous processing: every layer has a thread pool size and a queue. We can make some of those layers elastic, and try and adjust them according to how much work they do. But at some point it becomes a burden, and we start looking for something simpler, or leaner. Analysis of scalability leads to the conclusion that it is often better to shed the extra threads, and work with the constraints imposed by the physical hardware. This is an example of "mechanical sympathy", as is famously exploited by LMAX to great effect in the [Disruptor Pattern](https://lmax-exchange.github.io/disruptor/).

We have begun to see the power of the Reactive approach, but remember that with power comes responsibility. It’s radical, and it’s fundamental. It’s "rip it up and start again" territory. So you will also hopefully appreciate that Reactive isn’t a solution to all problems. In fact it isn’t a solution to any problem, it merely facilitates the solution of a certain class of problems. The benefits you get from using it might be outweighed by the costs of learning it, modifying your APIs to be Reactive all the way down, and maintaining the code afterwards, so tread carefully.

[**Reactive Programming in the Netflix API with RxJava**](https://medium.com/netflix-techblog/reactive-programming-in-the-netflix-api-with-rxjava-7811c3a1496a)

*by* [*Ben Christensen*](https://twitter.com/benjchristensen/) *and* [*Jafar Husain*](https://twitter.com/jhusain)

Our recent post on [optimizing the Netflix API](https://medium.com/@Netflix_Techblog/optimizing-the-netflix-api-5c9ac715cf19) introduced how our web service endpoints are implemented using a reactive programming model for composition of asynchronous callbacks from our service layer.

[**Optimizing the Netflix API**  
*how we redesigned our API to help UI teams meet their needs*medium.com](https://medium.com/@Netflix_Techblog/optimizing-the-netflix-api-5c9ac715cf19)

This post takes a closer look at how and why we use the reactive model and introduces our open source project RxJava — a Java implementation of [Rx (Reactive Extensions)](https://rx.codeplex.com/).

**Embrace Concurrency**

Server-side concurrency is needed to effectively reduce network chattiness. Without concurrent execution on the server, a single “heavy” client request might not be much better than many “light” requests because each network request from a device naturally executes in parallel with other network requests. If the server-side execution of a collapsed “heavy” request does not achieve a similar level of parallel execution it may be slower than the multiple “light” requests even accounting for saved network latency.

**Java Futures are Expensive to Compose**

Java [Futures](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html) are straight-forward to use for a [single level](https://gist.github.com/4670979) of asynchronous execution but they start to add non-trivial complexity when they’re [nested](https://gist.github.com/4671081) (prior to Java 8 CompletableFuture).

Conditional asynchronous execution flows become [difficult](https://gist.github.com/4671081#file-futuresb-java-L163) to optimally compose (particularly as latencies of each request vary at runtime) using Futures. It [can be done](http://www.amazon.com/gp/product/0321349601?ie=UTF8&tag=none0b69&linkCode=as2&camp=1789&creative=9325&creativeASIN=0321349601) of course, but it quickly becomes complicated (and thus error prone) or prematurely blocks on ‘Future.get()’, eliminating the benefit of asynchronous execution.

**Callbacks Have Their Own Problems**

Callbacks offer a solution to the tendency to block on Future.get() by not allowing anything to block. They are naturally efficient because they execute when the response is ready.

Similar to Futures though, they are easy to use with a single level of asynchronous execution but become [unwieldy](https://gist.github.com/4677544) with nested composition.

**Reactive**

Reactive programming offers efficient execution and composition by providing a collection of operators capable of filtering, selecting, transforming, combining and composing Observables.

The Observable data type can be thought of as a “push” equivalent to [Iterable](http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html) which is “pull”. With an Iterable, the consumer pulls values from the producer and the thread blocks until those values arrive. By contrast with the Observable type, the producer pushes values to the consumer whenever values are available. This approach is more flexible, because values can arrive synchronously or asynchronously.

The Observable type adds two missing semantics to the Gang of Four’s [Observer](http://en.wikipedia.org/wiki/Observer_pattern) pattern, which are available in the Iterable type:

1. The ability for the producer to signal to the consumer that there is no more data available.
2. The ability for the producer to signal to the consumer that an error has occurred.

With these two simple additions, we have unified the Iterable and Observable types. The only difference between them is the direction in which the data flows. This is very important because now any operation we perform on an Iterable, can also be performed on an Observable. Let’s take a look at an example…

/\*\*  
 \* Asynchronously calls 'customObservableNonBlocking' and defines   
 \* a chain of operators to apply to the callback sequence.  
 \*/  
def simpleComposition() {  
 // fetch an asynchronous Observable<String>  
 // that emits 75 Strings of 'anotherValue\_#'  
 customObservableNonBlocking()  
 // skip the first 10  
 .skip(10)  
 // take the next 5  
 .take(5)  
 // transform each String with the provided function  
 .map({ stringValue -> return stringValue + "\_transformed"})  
 // subscribe to the sequence and print each transformed String  
 .subscribe({ println "onNext => " + it})  
 }

// output:  
onNext => anotherValue\_10\_transformed  
onNext => anotherValue\_11\_transformed  
onNext => anotherValue\_12\_transformed  
onNext => anotherValue\_13\_transformed  
onNext => anotherValue\_14\_transformed

**Observable Service Layer**

The Netflix API takes advantage of Rx by making the entire service layer asynchronous (or at least appear so) — all “service” methods return an Observable<T>.

Making all return types Observable combined with a functional programming style frees up the service layer implementation to safely use concurrency. It also enables the service layer implementation to:

* conditionally return immediately from a cache
* block instead of using threads if resources are constrained
* use multiple threads
* use non-blocking IO
* migrate an underlying implementation from network based to in-memory cache

This can all happen without ever changing how client code interacts with or composes responses.

In short, client code treats all interactions with the API as asynchronous but the implementation chooses if something is blocking or non-blocking.

This next example code demonstrates how a service layer method can choose whether to synchronously return data from an in-memory cache or asynchronously retrieve data from a remote service and callback with the data once retrieved. In both cases the client code consumes it the same way.

/\*\*  
 \* Non-blocking method that immediately returns the value  
 \* if available or uses a thread to fetch the value and   
 \* callback via `onNext()` when done.   
 \*/  
def Observable<T> getData(int id) {  
 if(availableInMemory) {  
 // if data available return immediately with data  
 return Observable.create({ observer ->  
 observer.onNext(valueFromMemory);  
 observer.onCompleted();  
 })  
 } else {  
 // else spawn thread or async IO to fetch data  
 return Observable.create({ observer ->  
 executor.submit({  
 try {  
 // do work on separate thread  
 T value = getValueFromRemoteService(id);  
 // callback with value  
 observer.onNext(value);  
 observer.onCompleted();  
 }catch(Exception e) {  
 observer.onError(e);  
 }  
 })  
 });  
 }  
}

Retaining this level of control in the service layer is a major architectural advantage particularly for maintaining and optimizing functionality over time. Many different endpoint implementations can be coded against an Observable API and they work efficiently and correctly with the current thread or one or more worker threads backing their execution.

The following code demonstrates the consumption of an Observable API with a common Netflix use case — a grid of movies:

/\*\*  
 \* Demonstrate how Rx is used to compose Observables together  
 \* such as how a web service would to generate a JSON response.  
 \*   
 \* The simulated methods for the metadata represent different   
 \* services that are often backed by network calls.  
 \*   
 \* This will return a sequence of dictionaries such as this:  
 \*   
 \* [id:1000, title:video-1000-title, length:5428, bookmark:0,   
 \* rating:[actual:4, average:3, predicted:0]]  
 \*/  
def Observable getVideoGridForDisplay(userId) {  
 getListOfLists(userId).mapMany({ VideoList list ->  
 // for each VideoList we want to fetch the videos  
 list.getVideos()  
 .take(10) // we only want the first 10 of each list  
 .mapMany({ Video video ->   
 // for each video we want to fetch metadata  
 def m = video.getMetadata().map({   
 Map<String, String> md ->   
 // transform to the data and format we want  
 return [title: md.get("title"),  
 length: md.get("duration")]  
 })  
 def b = video.getBookmark(userId).map({   
 position ->   
 return [bookmark: position]  
 })  
 def r = video.getRating(userId).map({   
 VideoRating rating ->   
 return [rating:   
 [actual: rating.getActualStarRating(),  
 average: rating.getAverageStarRating(),  
 predicted: rating.getPredictedStarRating()]]  
 })  
 // compose these together  
 return Observable.zip(m, b, r, {  
 metadata, bookmark, rating ->   
 // now transform to complete dictionary of data  
 // we want for each Video  
 return [id: video.videoId] << metadata << bookmark << rating  
 })  
 })   
 })  
}  
  
// emits results such as  
[id:1002, title:video-1002-title, length:5428, bookmark:0,   
 rating:[actual:2, average:4, predicted:3]]  
[id:1003, title:video-1003-title, length:5428, bookmark:0,   
 rating:[actual:4, average:4, predicted:4]]  
[id:1004, title:video-1004-title, length:5428, bookmark:0,   
 rating:[actual:4, average:1, predicted:1]]

That code is declarative and [lazy](http://en.wikipedia.org/wiki/Lazy_evaluation) as well as functionally “pure” in that no mutation of state is occurring that would cause thread-safety issues.

The API Service Layer is now free to change the behavior of the methods ‘getListOfLists’, ‘getVideos’, ‘getMetadata’, ‘getBookmark’ and ‘getRating’ — some blocking others non-blocking but all consumed the same way.

In the example, ‘getListOfLists’ pushes each ‘VideoList’ object via ‘onNext()’ and then ‘getVideos()’ operates on that same parent thread. The implementation of that method could however change from blocking to non-blocking and the code would not need to change.

**RxJava**

RxJava is our implementation of Rx for the JVM and is available in the [ReactiveX repository in Github](https://github.com/ReactiveX/RxJava) *(prior to September 2014 was in the Netflix repo)*.

It is not yet feature complete with the .Net version of Rx, but what is implemented has been in use for the past year in production within the Netflix API.

We are open sourcing the code as version 0.5 as a way to acknowledgement that it’s not yet feature complete. The outstanding work is logged in the [RxJava Issues](https://github.com/Netflix/RxJava/issues?milestone=1&state=open).

*(Update: As of August 2014 the project hit the* [*1.0.0 Release Candidate*](https://github.com/ReactiveX/RxJava/releases/tag/v1.0.0-rc.1) *milestone.)*

Documentation is available on the [RxJava Wiki](https://github.com/ReactiveX/RxJava/wiki) including links to material available on the internet.

Some of the goals of RxJava are:

* Stay close to the original Rx.Net implementation while adjusting naming conventions and idioms to Java
* All contracts of Rx should be the same
* Target the JVM not a language. The first languages supported (beyond Java itself) are [Groovy](https://github.com/ReactiveX/RxGroovy), [Clojure](https://github.com/ReactiveX/RxClojure), [Scala](https://github.com/ReactiveX/RxScala) and [JRuby](https://github.com/ReactiveX/RxJRuby). New language adapters can be [contributed](https://github.com/ReactiveX/RxJava/wiki/How-to-Contribute).
* Support Java 6 (to include Android support) and higher with an eventual goal to target a build for Java 8 with its lambda support. *(Update: Java 8 support was achieved without a separate build)*

Here is an implementation of one of the examples above but using Clojure instead of Groovy:

(defn simpleComposition []  
 "Asynchronously calls 'customObservableNonBlocking' and defines a   
 chain of operators to apply to the callback sequence."  
 (->   
 ; fetch an asynchronous Observable<String>   
 ; that emits 75 Strings of 'anotherValue\_#'  
 (customObservableNonBlocking)  
 ; skip the first 10  
 (.skip 10)  
 ; take the next 5  
 (.take 5)  
 ; transform each String with the provided function  
 (.map #(str % "\_transformed"))  
 ; subscribe to the sequence and print each transformed String  
 (.subscribe #(println "onNext =>" %))))  
  
; output  
onNext => anotherValue\_10\_transformed  
onNext => anotherValue\_11\_transformed  
onNext => anotherValue\_12\_transformed  
onNext => anotherValue\_13\_transformed  
onNext => anotherValue\_14\_transformed

**Summary**

Reactive programming with RxJava has enabled Netflix developers to leverage server-side concurrency without the typical thread-safety and synchronization concerns. The API service layer implementation has control over concurrency primitives, which enables us to pursue system performance improvements without fear of breaking client code.

RxJava is effective on the server for us and it spreads deeper into our code the more we use it.

We hope you find the RxJava project as useful as we have and look forward to your contributions.

If this type of work interests you we are [always looking](http://jobs.netflix.com/) for talented engineers.

**September 2014 Update**

* This blog post originally used the term “functional reactive programming” or FRP. This term was used in error. RxJava does not implement “continuous time” which is a requirement for FRP from previous literature.
* Updated to new ReactiveX location for RxJava.

*Originally published at* [*techblog.netflix.com*](http://techblog.netflix.com/2013/02/rxjava-netflix-api.html) *on February 4, 2013.*

# [Advanced Reactive Java](https://akarnokd.blogspot.nl/)

### [RxJava design retrospect](https://akarnokd.blogspot.nl/2016/03/rxjava-design-retrospect.html)

### Intoduction

RxJava is now out more than 3 years and lived through several significant version changes. In this blog post, I'll point out design and implementation decisions that I personally think wasn't such a good idea.  
  
Don't get me wrong, it doesn't mean that RxJava is bad or I knew all along how to do it "properly". It was a learning process for all of us involved, but the question is, can we learn from those mistakes and do it better in the next major version?

### Synchronous unsubscription

In the early days, RxJava mirrored the architecture of Rx.NET which consisted of two important interfaces, **IObservable** and **IObserver**, derived through dualizing the **IEnumerable** and **IEnumerator**. (This was also true for my own library, Reactive4Java).  
  
If we look at **IObservable**, we find the **subscribe()** method that returns an **IDisposable**. This returned object allows one to dispose or cancel a running sequence. However, it has a critical problem I demonstrate with a minimalistic reactive program:

interface IDisposable {

    void dispose();

}

interface IObserver<T> {

   void onNext(T t);

}

interface IObservable<T> {

    IDisposable subscribe(IObserver<T> observer);

}

IObservable<Integer> source = o -> {

   for (int i = 0; i < Integer.MAX\_VALUE; i++) {

       o.onNext(i);

   }

   return () -> { };

};

IDisposable d = o.subscribe(System.out::println);

d.dispose();

If we run this code, it starts to print a lot of numbers to the console, despite we called dispose on the returned object by the subscribe method. What's wrong?  
  
The problem is that the source observable can only return its **IDisposable** object only after the for-loop finishes, but then it has nothing to do. The whole setup is synchronous and thus this structure can't be reasonably cancelled.  
  
Although Rx is good at async processing, many steps in a typical pipeline is synchronous and is affected by this synchronous cancellation requirement. Since Rx.NET is at least 3 years older than RxJava, how could this shortcoming still be in today's Rx.NET?  
  
The example code above is just the well known **range()** operator and if we run a similar code in C#, we find that it doesn't print or stops printing almost immediately. The secret is that Rx.NET's **range()** operator runs on async scheduler by default, so that for loop runs on a different thread and the operator can immediately return with a meaningful **IDisposable**. Therefore, the synchronous cancellation issue is averted, but I wonder, was it a conscious or unconscious decision to sidestep the underlying problem? Who knows.  
  
If you look at the source code of Rx.NET's range, you'll find something more complicated. It uses a recursive scheduling technique to deliver each value to the observer. When I measured it, it could only sustain 1M ops/second on the same machine which could do 250M ops/second with RxJava while delivering 1M elements.  
  
Now RxJava's **range()** never used a scheduler thus the synchronous cancellation problem was discovered and mitigated by introducing the **Subscriber** class. An instance can be checked if it still wants events or not. The example above can be rewritten so the for loop checks its subscriber and quits accordingly.

Observable<Integer> source = Observable.create(s -> {

   for (int i = 0; i < Integer.MAX\_VALUE && !s.isUnsubscribed(); i++) {

       o.onNext(i);

   }

});

Subscription d = o.subscribe(new Subscriber<Integer>() {

@Override

public void onNext(Integer v) {

System.out.println(v);

unsubscribe();

}

// ...

});

The Subscriber acts like a **take(1)** and unsubscribes itself after the first item. This **unsubscribe()** call internally sets a volatile boolean flag that is read by **isUnsubscribed()** and the loop above is stopped. Note, however, that you still can't unsubscribe via the **Subscription** returned by **subscribe()** because the lambda with the for loop doesn't exit until its terminal condition is met.  
  
It doesn't seem to solve our initial problem that well, does it? Luckily, this new structure has the property that it can be cancelled before or while the loop is running, the latter done from another thread of course:

Subscriber<Integer> s = new Subscriber<Integer>() {

@Override

public void onNext(Integer v) {

System.out.println();

}

// ...

}

Schedulers.computation().schedule(s::unsubscribe, 1, TimeUnit.SECONDS);

source.subscribe(s);

The fact that you can basically inject the cancellation support upfront, before anything is even subscribed to, allows proper propagation of unsubscription even with the most complicated operators.  
  
In addition, there is a deeper implication of this new structure. In the new **Observable**, the lambda doesn't return anything, but **Observable.subscribe()** does still return a **Subscription** which is practically the same **Subscriber** sent in as the parameter. (Technically, it is a bit more involved process; see Jake Wharton's excellent [video talk](http://jakewharton.com/presentation/2015-11-05-oredev/) on the subject).  
  
The insight: you can't be fully reactive if you return something. Returning something implies synchronous behavior and the method has to provide some result, even though it can't at that moment. This is when one is forced to block or sleep until the real logic can produce the relevant object to be returned. I showed this in [my post](http://akarnokd.blogspot.hu/2015/11/asynchronous-event-streams-vs-reactive.html) about the OSGi Asynchronous Event Streams initiative.

### Resources of the Subscriber

The **Subscriber** class offers the ability to associate resources with it in the form of **Subscription** instances. When the operator is cancelled (or terminates), these resources are unsubscribed as well.  
  
This is quite a convenience for operator developers, however, has its own cost: allocation.  
  
Whenever a **Subscriber** is instantiated with the default constructor, an inner **SubscriptionList** instance is also always created, whether or not the Subscriber is likely to hold resources or not. In the previous example, **range()** doesn't need resources thus the **SubscriptionList** is never really used.  
  
On one hand, there exist many operators that don't manage resources so creating the extra container is wasteful. On the other hand, many operators do use resources and expect this convenience to be present.  
  
In addition, you may recall that **Subscriber** has a constructor that takes another **Subscriber** and gives the option to share the underlying **SubscriptionList**. Certainly, this could help reduce the allocation count, but most operators that use resources themselves can't share the same underlying **SubscriptionList** as this would allow them to unsubscribe resources downstream (see [pitfall #2](http://akarnokd.blogspot.hu/2015/05/pitfalls-of-operator-implementations.html)). Thus, the current **Subscriber** structure is more of a burden, performance wise, than a win for operator writers.  
  
You may now think, what's wrong with giving convenience tools to operator writers? I agree that operators implemented outside RxJava should get as much help as reasonable, however, I believe internal operators should take the diligence and have efficient implementations to begin with.  
  
I've tried a few times to resolve this problem but given the architecture of 1.x, I have doubts it can be achieved. Fortunately, the Reactive-Streams' architecture and thus RxJava 2.x solves this problem by making the resource management the responsibility of the operators.

### Subscriber request

If you look into how **Subscriber** is implemented, you'll see the protected final **request()** method. This is a convenience method that makes sure if there is a **Producer** set via **setProducer**, the request is forwarded to it or accumulated until one **Producer** arrives. Basically, this is an inlined [producer-arbiter](http://akarnokd.blogspot.hu/2015/05/operator-concurrency-primitives_15.html).  
  
One might think the method's implementation gives significant overhead to request management, but JMH benchmarks confirmed they don't really affect the overhead outside a small +/- 3% difference, that may also be due to noise.  
  
The real problem with this method is that it has the same name as **Producer.request**, making it impossible to implement **Producer** when one extends **Subscriber** at the same time.  
  
This has the unfortunate consequence that one usually needs an extra **Producer** object along with the main **Subscriber** if the operator does some request-manipulation.  
  
This has the consequence of extra allocation during subscription time which affects GC the most with short-lived sequences. The second property is that it increases the call-stack depth and may prevent some JIT optimization.  
  
Since **Subscriber.request()** is also part of the public API, it can't be renamed in 1.x to make room for **Producer.request()**.  
  
Again, the solution will come with 2.x: there, since Reactive-Streams **Subscriber** and **Subscription** are both interfaces, both can appear the same time, plus, a convenience **request()** method can be moved into a convenience implementation of **Subscriber** (i.e., **AsyncSubscriber**) without affecting the operator internals. (This also means it will be discouraged to use convenience **Subscriber**s within operators.)

### Lift

Along with backpressure, the method **Observable.lift()** is considered by many as the best addition to the library. It lets you step into the subscription process and given a Subscriber from downstream, you can return another **Subscriber** for upstream that does the business logic for that operator.  
  
It became so popular almost all instance operators of **Observable** are now using it.  
  
Unfortunately, the convenience has a cost: allocation. For most operators, applying that operator to a sequence incurs 3 object allocations. To show this, let's unroll the application of the **map()** operator:

public final <R> Observable<R> map(Func1<? super T, ? extends R> func) {

OperatorMap<T, R> op = new OperatorMap<T, R>(func);

return new Observable<R>(new OnSubscribe<R>() {

@Override

public void call(Subscriber<? super R> child) {

Subscriber<? super T> parent = op.call(child);

Observable.this.unsafeSubscribe(parent);

}

});

}

We have 1) the **Operator** instance, 2) the **Observable** instance and 3) the **OnSubscribe** instance for each application.  
  
This may not of concern for direct sequences that use map, but imagine you have this 3 allocation a million times because you happen to **flatMap** something whose inner **Observable**s have operators applied to them:

Observable.range(1, 1\_000\_000).flatMap(v ->

Observable.just(v).observeOn(Schedulers.computation()).map(v -> v \* v))

.subscribe(...);

The **lift** operator is practically an **OnSubscribe** instance that captures the upstream **Observable** and calls **Operator.call** with the downstream **Subscriber**. Clearly, one could just implement operators directly with **OnSubscribe** and have the upstream **Observable** as a parameter; the total instance sizes wouldn't change much but both the allocation count and stack depth get reduced.  
  
The current lift structure has another adverse effect: it makes operator-fusion difficult to impossible in its current form because 1) it is an anonymous class and one can't discover its upstream **Observable** and **Operator** easily, and 2) even if made a named class, the two classes are hidden behind indirection and any discovery process now faces more overhead.  
  
Luckily, the shortcomings mentioned so far can be remedied without affecting the public API, but requires diligence of writing and reviewing thousands of lines of code changes.  
  
Unfortunately, when I implemented RxJava 2.0 developer-preview last September, I did not think of this overhead thus the current 2.x branch still uses **lift()** extensively.  
  
However, there is light at the end of the tunnel: Project Reactor 2.5 doesn't go down on the **lift()** path and now has lower overhead than RxJava.

### Create

Lately, I'm quite outspoken against **Observable.create()** and now I think it should be named something more scary so beginners avoid it and look for proper factory methods in **Observable** that do backpressure and unsubscription properly. I can see it as a tool for demonstating to one's audience how to enter the reactive world, but I'm convinced it should receive less spotlight in those presentations.  
  
Regardless, the problem with **create()** is that it encourages creating 2 instances per **Observable**: 1) the **Observable** instance itself and the **OnSubscribe** holding the subscription logic.  
  
The approach that one creates an **Observable** instance with **create()** was born from the encouragement: "composition over inheritance". From general design perspective, this sounds okay, but one has to note that in Java, composition means object allocation: outer objects and inner objects, and more inner-inner objects.  
  
To avoid all these allocations, the solution would be to make **Observable** not hold an instance of **OnSubscribe** by default (but keep **create()** as the lambda-factory version) and operators (both source and intermediate) should extend **Observable** directly. All operator methods would still reside in **Observable**:

public final <R> Observable<R> map(Func1<? super T, ? extends R> func) {

return new ObservableMap<T, R>(this, func);

}

Thus, without **lift()** and **create()**, **map()** would allocate a single **Observable** instance per application.  
  
Such change, I believe, wouldn't affect the public API since **Observable** methods are static or final to begin with and operators would be still a subclass of **Observable**. The change also would help with operator-fusion because each upstream source can now be directly identified and its parameters exposed without indirection.  
  
Again, Project Reactor 2.5 is ahead of RxJava and doesn't use the **create()** mechanics. Its operators are implemented extending a base class, [**Flux**](https://github.com/reactor/reactor-core/blob/master/src/main/java/reactor/core/publisher/Flux.java), the way suggested above.

### Conclusion

Designing and implementing RxJava version was and is a learning process as well with unanticipated effects on complexity and performance.  
  
You may think, why the hassle about structures and allocations that clearly work in their current form? Two reasons: the Cloud and Android/IoT. For the cloud, where billions of events happen, any inefficiency or unnecessary overhead is amplified along with the numbers. You may not easily calculate how much does that range-flatmap example above cost you on your laptop, but Cloud suppliers will make you pay for each second, gigabyte and gigahertz of using their service. For Android and IoT, the resource constraints of the devices and the expectancy of more and more features requires one - eventually - to budget memory usage, GC and battery life.

### [Operator-fusion (Part 1)](https://akarnokd.blogspot.nl/2016/03/operator-fusion-part-1.html)

### Introduction

Operator-fusion, one of the cutting-edge research topics in the reactive programming world, is the aim to have two of more subsequent operators combined in a way that reduces overhead (time, memory) of the dataflow.  
  
*(Other cutting-edge topics are: 1) reactive IO, 2) more native parallel async sequences and 3) transparent remote queries.)*  
  
The key insight with operator-fusion is threefold:

1. many sequences are started from constant or quasi-constant sources such as **just()**, **from(T[])**, **from(Iterable)**, **fromCallable()** which don't really need the thread-safety dance in a sequence of operators,
2. some pairs of operators can share internal components such as **Queue**s and
3. some operators can tell if they consumed the value or dropped it, avoiding **request(1)** call overhead.

In this mini-series, I'll describe the hows and whys of operator-fusion, as we currently understand it. By "we", I mean the joint research effort on optimizing Reactive-Streams operators beyond what's there in RxJava 2.x and has been in previous versions of Project Reactor.  
  
The experimentation happens in the [reactive-streams-commons](https://github.com/reactor/reactive-streams-commons), **Rsc** for short, GitHub repository. The results of the Rsc is now driving Project Reactor 2.5 (currently in milestone 2) and verified by a large user base. Hopefully, RxJava can benefit from the results as well (but maybe not before 3.x).  
  
If you are following Akka-Streams, you might have read/head about operator-fusion there as well. As far as I could [understand their approach](http://doc.akka.io/docs/akka/snapshot/java/stream/stream-flows-and-basics.html#Operator_Fusion), their objective is to make sure more stages of the pipeline run on the same Actor, avoiding the previous, very likely, thread-hopping with their sequences. Essentially, there is now a mode where the developer can define the async boundaries in the pipeline. Does this sound familiar? From day 1, Rx-based libraries let you do this.

### Generations

Reactive libraries and associated concepts evolved over time. What we had 7 years ago in Rx.NET, requirements and implementation-wise is significantly different what we'll have tomorrow with libraries such as Project Reactor.  
  
With my experience with the history of "modern" reactive programming, I categorize the libraries into generations.

#### 0th generation

The very first generation of reactive programming tools mainly consist of **java.util.Observable** API and its cousins in other languages and almost any callback-based API such as **addXXXListener** in Swing/AWT/Android.  
  
The Observable API was most likely derived from the Gang-of-four design patterns book (or the other way around, who knows) and has the drawback of being inconvenient to use and non-composable. In today's terms, it is a limited **PublishSubject** where you have only one stage: publisher-subscriber.  
  
The  **addXXXListener** style of APIs suffer, although facilitate push-based eventing, from composability deficiencies. The lack of common base concept would require you to implement a composable library for each of them one-by-one; or have one common abstraction like RxJava and build adapter for each **addXXXListener/removeXXXListener** entry point.

#### 1st generation

Once the deficiencies were recognized and addressed by Erik Meijer & Team at Microsoft, the first generation of reactive programming libraries were born: Rx.NET around 2010, Reactive4Java in 2011 and early versions of RxJava in 2013.  
  
The others followed the Rx.NET architecture closely, but soon turned out there are problems with this architecture. When the original **IObservable/IObserver** is implemented with purely same-thread manner, the sequences can't be cancelled in progress with operators such as **take()**. Rx.NET sidestepped the issue by using mandatory asnycrony in sources such as **range()**.  
  
The second problem was the case when the producer side is separated by an implicit or explicit asynchronous boundary from a consumer that can't do its job fast enough. This can happen with trivial consumers as well because of the infrastructure overhead of crossing the asynchronous boundary. This is what we call the backpressure problem.

#### 2nd generation

The new deficiencies of synchronous cancellation and the lack of backpressure were recognized by the RxJava team (I wasn't really involved) and a new architecture has been designed.  
  
The class **Subscriber** was introduced which could tell if it was interested in more events or not via **isUnsubscribed()** that had to be checked by each source or operator emitting events.  
  
The backpressure problem was addressed by using co-routines to signal the amount of items a Subscriber can process at a time through a **Producer** interface.  
  
The third addition was the method **lift()** which allows a functional transformation between **Subscriber**s directly. Almost all instance operators have been rewritten to run with **lift()** through the new **Operator** interface.

#### 3rd generation

Apart from being [clumsy and limiting some optimizations](http://akarnokd.blogspot.hu/2016/03/rxjava-design-retrospect.html), the problem with RxJava's solution was that it was incompatible with the viewpoints of other (upcoming) reactive libraries at the time. Recognizing the advent of (backpressure enabled) reactive programming, engineers from various companies got together and created the Reactive-Streams specification. The main output is a set of 4 interfaces and 30 rules regarding them and their 7 total methods.  
  
The Reactive-Streams specification allows library implementors to be compatible with each other and compose the sequences, cancellation and backpressure across library boundaries while allowing the end-user to switch between implementations at will.  
  
Reactive-Streams, and thus 3rd generation, libraries are, for example, RxJava 2.x, Project Reactor and Akka-Streams.

#### 4th generation

Implementing a fluent library on top of Reactive-Streams requires quite a different internal architecture, thus RxJava 2.x had to be rewritten from scratch. While I was doing this reimplementation, I recognized some operators could be combined in an external or internal fashion, saving on various overheads such as queueing, concurrency-atomics and requesting more.  
  
Since RxJava 2.x development crawled to halt due to lack of serious interest from certain parties, I set RxJava 2.x aside until Stephane Maldini (one of the contributors to Reactive-Streams and main contributor to Project Reactor) and I started talking about a set of foundational operators that both RxJava 2.x and Project Reactor 2.5+ (and eventually Akka-Streams) could use and incorporate them into the respective libraries.  
  
With active communication, we established the reactive-streams-commons library, built the foundational operators and designed the components of optimizations that we call now **operator-fusion**.  
  
Thus, a 4th generation reactive library may look like a 3rd generation from the outside, but the internals of many operators change significantly to support overhead reduction even further.

#### 5+ generation

I think, at this point, we are at half point in what operator-fusion can achieve, but there are signs the architecture of Reactive-Streams will need extensions to support reactive IO operations in the form of bi-directional sequences (or channels). In addition, transparent remote reactive queries may require changes as well (see QBservable in Rx.NET). I don't see the full extent of possibilities and requirements at this point and all is open for discussion.

### The Rx lifecycle

Before jumping into operator-fusion, I'd like to define the major points (thus the terminology I'll be using) of the lifecycle of an Rx sequence. This applies to any version of RxJava and any Reactive-Streams based libraries as well.  
  
The lifecycle can be split into 3 main points:

1. **Assembly-time**. This is the time when you write up **just().subscribeOn().map()** and assign that to a field or variable of type **Observable/Publisher**. This is the main difference between Future-based APIs (**Promise**, **CompletableFuture**, etc.) which if support some fluent API where there isn't a separate assembly time but some form of interleaving among the 3 points.
2. **Subscription-time**. This is the time when a Subscriber subscribes to a sequence at its very end and triggers a "storm" of subscriptions inside the various operators. On one hand, it has an upstream-directed edge and on the other hand, a downstream-directed edge of calls to **setProducer**/**onSubscribe**. This is when subscription-sideeffects are triggered and generally no value is flowing through the pipeline.
3. **Runtime**. This is the time when items are generated followed by zero or one terminal event of error/completion.

Each distinct point in the lifecycle enables a different set of optimization possibilities.

### Operator-fusion

I admit, I took the term operator-fusion from some Intel CPU documentation describing their internal architecture doing macro- and micro-fusions on assembly-level operators. It kinda sounded cool and the concepts behind it could be expanded up the language level and reach the operators of reactive dataflows.  
  
The idea, on the reactive level, is to modify the sequence the user created at various lifecycle points to remove overhead mandated by the general architecture of the reactive library.  
  
As with the assembly-level fusion, we can define two kinds of reactive operator-fusion.

#### Macro-fusion

Macro-fusion happens mainly in the assembly-time in the form of replacing two or more subsequent operators with a single operator, thus reducing the subscription-time overhead (and sometimes the runtime overhead in case the JIT would be overwhelmed) of the sequence. There are several ways this can happen.  
  
**1) Replacing an operator with another operator**  
  
In this form of fusion, the operator applied looks at the upstream source (this is why I [mentioned](http://akarnokd.blogspot.hu/2016/03/rxjava-design-retrospect.html) **lift()** causes trouble) and instead of instantiating its own implementation, it calls/instantiates a different operator.  
  
One example of this is when you try to **amb()/concat()/merge()** an array of sources which has only one element. In this case, it would be unnecessary to instantiate the implementation and one can avoid the overhead by returning that single element directly. This kind of optimization is already part of RxJava 1.x.  
  
The second example is when one uses a constant source, such as **range()** and applies **subscribeOn()**. However, there is little-to-no behavioral difference between applying **observeOn()** in the same situation. Thus **subscribeOn()** detecting a **range()** can switch to **observeOn()** and perhaps benefit from other optimizations that **observeOn()** itself can provide.  
  
**2) Replacing an operator with a custom operator**  
  
The exist operator-pairs that come up often and may work better if they were combined into a single operator. A very common operator-pair that is used for jump-starting some asynchronous computation is **just().subscribeOn()** or the equivalent **just().observeOn()**.  
  
Such sequences have quite a large overhead compared to the single value they emit: internal queues get created, workers get instantiated and released, several atomic variables are modified.  
  
Therefore, replacing the pair with a custom operator that combines the scheduling and emission into a single value into one single operator is a win.  
  
This approach, especially involving **just()**, can be extended to other operators, such as **flatMap()** where all the [internal complexities](http://akarnokd.blogspot.hu/2016/02/flatmap-part-1.html) can be avoided by invoking the mapper function once and running with the single **Observable/Publisher** directly, without buffering or extra synchronization.  
  
Again, RxJava 1.x already has optimizations such as these examples above.  
  
**3) Replacing during subscription-time**  
  
There are cases when the previous two cases may happen during subscription-time instead of assembly-time.  
  
I can see two reasons for moving the optimization into the subscription-time: 1) safety-net in case the fluent API is bypassed and 2) convenience if the fused and non-fused version doesn't differ that much to warrant a full-independed class as operator.  
  
**4) Replacing with the same operator but with modified parameters**  
  
Users of the libraries tend to apply certain operator types multiple times in a sequence, such as **map()** and **filter()**:

Observable.range(1, 10)

   .filter(v -> v % 3 == 0)

   .filter(v -> v % 2 == 0)

   .map(v -> v + 1)

   .map(v -> v \* v)

   .subscribe(System.out::println);

This is quite convenient to look at one can more easily understand what's happening. Unfortunately, if you have a range of 1M or resubscribe to the sequence a million times, the structure has quite a measurable overhead compared to a flatter structure.  
  
The idea with this macro-fusion is to detect if an operator of the same type was applied before, take the original source and apply the operator where the parameters get combined. In our example, that means **range()** is followed, internally, by a single **filter()** application where the two lambda functions (in their reference form) are combined:

Predicate<Integer> p1 = v -> v % 3 == 0;

Predicate<Integer> p2 = v -> v % 2 == 0;

Predicate<Integer> p3 = v -> p1.test(v) && p2.test(v);

A similar fusion happens with the lambda of the **map()** operations, with the difference that the output of the first lambda is going to be the input of the second lambda:

Function<Integer, Integer> f1 = v -> v + 1;

Function<Integer, Integer> f2 = v -> v \* v;

Function<Integer, Integer> f3 = v -> f2.apply(f1.apply(v));

#### Micro-fusion

Micro-fusion happens when two or more operators share their resources or internal structures and thus bypassing some overhead of the general wired-up structure. Micro-fusion can mostly happen in subscription-time.  
  
The original idea of micro-fusion was the recognition that operators that end in an output queue and operators starting with a front-queue could share the same **Queue** instance, saving on allocation and saving on the drain-loop work-in-progress serialization atomics. Later, the concept has been extended to sources that could pose as **Queue**s and thus avoiding the creation of **SpscArrayQueue** instances completely.  
  
There are several forms of micro-fusion that can happen in operators.  
  
**1) Conditional Subscriber**  
  
When filtering an (upstream) source with **filter()** or **distinct()**, if that source features a drain-loop with request accounting, there is the likely scenario that **filter()** will **request(1)** if the last value has been dropped by the operator. Lots of **request(1)** calls, which all trigger some atomic increment or CAS loop adds up overhead relatively quickly.  
  
The idea behind a conditional subscriber is to have an extra method, **boolean onNextIf(T v)**, that would indicate if it didn't really consume the value. In that case, the usual drain-loop would then skip incrementing its emission counter and keep emitting until the request limit is reached by successful consumptions.  
  
This saves a lot on request management overhead and some operators in RxJava 2.x support it, but there are some drawbacks as well, mostly affecting the library writers themselves:  
  
a) The source and filter may be separated by other operators so those operators have to offer a conditional **Subscriber** version of themselves to pass along the **onNextIf()** calls.  
  
b) By returning non-void, the **onNextIf()** implementation is forced to be synchronous in nature. However, since it just returns a **boolean**, it can still behave as the regular **onNext()** method by claiming it consumed the value even though it dropped it; therefore, it has to **request(1)** manually again.  
  
Since this is an internal affair, conditional **Subscriber**s of operators still have to implement the regular **onNext()** behavior in case the upstream doesn't support conditional emission and/or is from some other reactive library with different internals.  
  
**2) Synchronous-fusion**  
  
We call synchronous micro-fusion the cases when the source to an operator is synchronous in nature, and can pretend to be a **Queue** itself.  
  
Typical sources of such nature are **range()**, **fromIterable**, **fromArray**, **fromStream** and **fromCallable**. You could count **just()** here as well but usually, it is involved more in macro-fusion cases.  
  
Operators that use an internal queues are, for example, **observeOn()**, **flatMap()** in its [inner sources](http://akarnokd.blogspot.hu/2016/03/flatmap-part-2.html), **publish()**, **zip()**, etc.  
  
The idea is for the source's **Subscription** to also implement **Queue**, and during the subscription time, the **onSubscribe()** can check for it and use it instead of newing up its internal **Queue** implementation.  
  
This requires a different operation mode (a mode switch) from both upstream and the operator itself, namely, calling **request()** is forbidded and one has to remember the mode itself in some field variable. In addition, when the **Queue.poll()** returns null, that should indicate no more values will ever come, unlike regular **poll()**s in operators where null means no values available but there could be in the future.  
  
Unfortunately for the RxJava 1.x, this fusion works better with the Reactive-Streams architecture because a) setting a **Producer** is optional, b) the lifecycle-related behaviors are too unreliable and c) discovery difficulties and too much indirection.  
  
When benchmarked in Rsc, this form of fusion makes a **range().observeOn()** sequence go from 55M Ops/s to 200M Ops/s in throughput, giving a ~4x overhead reduction in this trivial sequence.  
  
Again, there are downsides of this kind of API "hacking":  
  
a) In short sequences, the mode switch inside the operator may not be worth it.  
  
b) This optimization is library local at the moment so unless there is a standard API like with Reactive-Streams interfaces, library A implementing micro-fusion may not cross-fuse with library B.  
  
c) There are situations where this queue-fusion optimization is invalid, mainly due to thread-boundary violations (or other effects we haven't discovered yet that create invalid fused sequences).  
  
d) This optimization has also some library-spanning effect, because intermediate operators have to support, or at least not interfere with the setup protocol of the mode-switch.  
  
e) This also has the effect that in a Reactive-Streams architecture, an operator can't just pass along the Subscription from upstream to its downstream because if they fuse, the intermediate operator is cut out.  
  
**3) Asynchronous-fusion**  
  
There are other situations when the source has its own internal, downstream facing queue which is drained by requests, but the timing and count of the items are not known upfront.  
  
In this situation, the source can also implement the **Queue** interface and the operator use it instead of a fresh queue, but the protocol has to change, especially if the same operator wants to support synchronous fusion.  
  
Therefore, in Rsc, instead of checking if **Subscription** implements **Queue** received in **onSubscribe()**, we established a custom interface, **QueueSubscription**, that implements **Subscription**, **Queue** and a method called **requestFusion()**.  
  
The method **requestFusion()** takes a **int**-flag telling the upstream what kind of fusion the the current operator wants or supports and the upstream should respond what kind of fusion mode it has activated.  
  
For example, **flatMap()** would request a synchronous fusion from the inner source which could answer with, sorry-no, yes-synchronous or instead-asynchronous mode and act according to them. Generally, one can "downgrade" from a synchronous mode to asynchronous or none, but one can't "upgrade" to synchronous mode from asychronous mode requests.  
  
In asynchronous-fusion mode, downstream has to still issue **request()** calls, but instead of enqueueing the value twice, the value gets generated into the shared queue and the upstream calls **onNext()** indicating its availability. The value of this call is irrelevant, we use **null** as a type-neutral value, and can trigger the usual **drain()** call directly.  
  
Since fusion happens in subscription time, there is too late to change the **Subscriber** instance itself, therefore, one needs a mode flag in the operator and do a conditional check for the fusion mode. Therefore, the same class can work with regular and fuseable sources alike.  
  
This is the point when the complexity rises 50% above the complexity of a classical backpressured operator and requires quite an in-detail knowledge of all the operators and their behavior in various situations.

#### Invalid fusions

Before one goes ahead and fuses every queue in every operation, a problem comes up in the form of invalid fusion.  
  
Operators tend to have some barriers associated with them. These are somewhat analogous to memory barriers and have a similar effect: 1) prevent certain reorderings and 2) prevent certain optimizations altogether.  
  
For example, mapping from **String** to **Integer** and then **Integer** to **Double** can't be reordered because of the type mismatch. Reordering a **filter()** with **map()** may be invalid when the map changes types or by introducing side-effects in map that would have been avoided because filter didn't let the causing value through in the first place.  
  
On one hand, these functional barriers mainly affect the macro-fusion operators and somewhat easier the detect and understand.  
  
On the other hand, when asynchrony is involved, in the form of a thread-jumping behavior provided by **observeOn()**, micro-fusion can become invalid.  
  
For example, if you have a sequence of

source.flatMap(u -> range(1, 2).map(v -> heavyComputation(v))

.observeOn(AndroidSchedulers.mainThread()))

.subscribe(...)

The inner sequence of **range-map-observeOn-flatMap** would have a single fused queue, where the **map()**'s behavior has been reordered to the output side of the shared queue, now executes the heavy computation on the main thread.  
  
*On a side note, classical* ***observeOn*** *can also drag the emission to its thread due to how backpressure triggers emission, thus in the example above, if you have a longer* ***range()****, the range's emission and so the* ***map()****'s computation would end up on the main thread anyway. This is why one needs* ***subscribeOn()****/****observeOn()*** *before map to ensure it runs on the correct thread.*  
  
This required a slight change to the protocol of the **requestFusion()** call by introducing a bit indicating if the caller (chain) acts as an asynchronous boundary, that is, the endpoint of the fused queue would be in another thread. Intermediate operators such as **map()** intercept this method all and simply respond with no-fusion.  
  
Finally, there might be a subscription-time related barrier as well that prevents reordering/optimization due to subscription side-effects. We are not sure of this yet but here are a few hands-on cases that requires further study:  
  
1) Is it valid to turn a **range().subscribeOn(s1).observeOn(s2)** chain, which I call strongly-pipelined sequence because of the forced thread-boundary switch by default, into a fused **range().observeOn(s2)**? The tail-emission pattern is the same, you get events on **Scheduler** s2, but now we've lost the strong pipelining effect.  
  
2) Subscribing to a **Subject** may take some in case there are lots of Subscribers there thus **subscribeOn()** may be a valid use to offset the overhead, but generally, there are no other side-effects happening when one subscribes to a **PublishSubject**. Is it valid to drop/replace **subscribeOn()** here?

### Conclusion

Operator-fusion is a great opportunity, but also a great responsibility, to reduce overhead in reactive dataflows, and sometimes, get pretty close (+50% overhead with Project Reactor 2.5 M1 instead of +200% overhead with RxJava 2.x) to a regular Java **Stream**s sequence's overhead while still supporting asynchronous (parts of) sequences with the same API (and similar internals).  
  
However, adding fusion to every operator over zealously may not worth it and one should focus on operators doing the heavy lifting in user's code most of the time: **flatMap()**, **observeOn()**, **zip()**, **just()**, **from()** etc. In addition, one could say every operator pair is macro-fuseable because a custom operator can be written for it, but then you now have a combinatorial explosion of operators that now have to interact with the regular operators and with each other.  
  
Of course, on the other side, there are operators that don't look like they could be (micro-) fused but may turn up fuseable after all. But instead of building a huge operator cross-fusion matrix, there might be a possibility to automatically discover which operators can be fused by modelling them and the sequences in some way and applying graph algorithms on the network - a topic for further research.  
  
Anyway, the in the next part, I'll dive deeper into how operator-fusion in Rsc has been implemented, but before that, I'd like to describe the in-depth technicalities and differences of **subscribeOn()** and **observeOn()** operators in an intermediate post for two reasons:  
  
1) I think showing how to implement them clears up the confusion around them because I learned about **subscribeOn()** and **observeOn()**the same in-depth technical way in the first place (and I was never confused).  
  
2) Knowing their structure and exact behavior helps in understanding the fusion-related changes applied to them later on.  
  
As for where you can play with this fusion thing (as an end-user), check out [Project Reactor 2.5](https://github.com/reactor/reactor-core), who have extensively (unit-) tested the solutions I have described in the post. Of course, since this is an ongoing research, the [Rsc project](https://github.com/reactor/reactive-streams-commons) itself welcomes feedback or tips on what operator combinations we should optimize for.

### [Operator fusion (part 2 - final)](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

### Introduction

In the previous part, I've introduced the concepts around operator fusion. In this post, I'll detail the API and protocols required make operator fusion happen.  
  
In its current form, operator fusion works between two subsequent operators and is based on the ability to identify each other and, in case of micro-fusion, switch to a different protocol than Reactive-Streams (RS) if both agree.

### Macro-fusion constructs

The primary targets of macro-fusion are the single element sources: **just()**, **empty()**, **fromCallable()**. Firing up the complete RS infrastructure for such single elements is quite expensive, but half of the API use in RxJava and Reactor come from these. Therefore, RxJava introduced **Single** and Reactor introduced **Mono** to help as much as possible and offer (ever increasingly) optimized operators on them.  
  
However, knowing a source will generate 0 or 1 element during assembly time is also a great help in regular **Observable** / **Flux** uses. In addition, knowing the source is also a constant helps inlining it in via some custom operator.

#### Creating 0 or 1 element synchronous sources

To indicate a source returns a single value, the [Reactive-Streams-Commons](https://github.com/reactor/reactive-streams-commons) (Rsc) project (and Reactor off it) established a contract:  
  
**If a Publisher implements java.util.concurrent.Callable, it is considered a 0 or 1 element source.**  
  
You can implement **Callable** and return a non-**null** value that can be computed synchronously. You can also return **null** which indicates an empty result. (Remember, RS doesn't allow **null** values over **onNext**.) The call to **call()** will happen during subscription time.

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class MySingleSource implements Publisher<Object>, Callable<Object> {      @Override      public void subscribe(Subscriber<? super Object> s) {          s.onSubscribe(new ScalarSubscription<>(s, System.currentTimeMillis()));      }        @Override      public Object call() throws Exception {          return System.currentTimeMillis();      }  } |

If the 0 or 1 element source is known to be constant, the source can be the subject of assembly time optimizations. For example, if it returns **null**, indicating emptiness (like **empty()**), there are only a handful of operators that can be applied to it (which don't work on items) and the assembly process can just return **empty()**.  
  
**We can extend Callable with a new interface ScalarCallable to indicate a 0 or 1 element constant source.**

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4 | public interface ScalarCallable<T> extends Callable<T> {      @Override      T call();  } |

By extending **Callable**, any use places who expects a 0 or 1 element dynamic source can work with a constant source. The reverse is not true; those expecting a constant source won't execute an arbitrary **Callable** (which could block or trigger side-effects) during assembly time:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class MyScalarSource implements Publisher<Object>, ScalarCallable<Object> {      @Override      public void subscribe(Subscriber<? super Object> s) {          s.onSubscribe(new ScalarSubscription<>(s, 1));      }        @Override      public Object call() {          return 1;      }  } |

Note that the **ScalarCallable** overrides the **call()** method and removes the throws Exception clause: scalar constants should not throw for one and consumers should not need to wrap the **call()** into a try-catch.

#### Consuming 0 or 1 element synchronous sources

Consuming **Callable** and **ScalarCallable** is a matter of **instanceof** checks performed either in subscription time or assembly time respectively, followed by the extraction of the single value through **call()**.  
  
For example, a macro-fusion on the operator **count()** could check for a scalar value and return a constant 0 for an empty or 1 for a single value:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public final Flux<Long> count() {      if (this instanceof ScalarCallable) {           T value = ((ScalarCallable<T>)this).call();           return just(value == null ? 0 : 1);      }      return new FluxCount<>(this);  } |

Another example is to have a shortcut in **flatMap()**, **concatMap()** or **switchMap()** for 0 or 1 element sources. In this case, there is no need to run the full infrastructure but just subscribe to the Publisher returned by their mapping function.  
  
Note that since the mapper function can side-effect itself, one can't use assembly-time optimization on them and a new source operator has to be introduced.

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public final <R> Px<R> flatMap(          Function<? super T, ? extends Publisher<? extends R>> mapper) {        if (this instanceof Callable) {            return new PublisherCallableMap<((Callable<T>)this, mapper);      }        return new PublisherFlatMap<>(this, mapper, ...);  } |

(Remark: **Px** stands for **Publisher Extensions** in Rsc and is the base type for Rsc's fluent API - more of a convenience in tests and perf to avoid spelling out all those **PubliserXXX** classes than a fully fledged API entry point.)

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | public final class PublisherCallableMap<T, R> implements Publisher<R> {      final Callable<? extends T> source;      final Function<? super T, ? extends Publisher<? extends T>> mapper;        public PublisherCallableMap(              Callable<? extends T> source,              Function<? super T, ? extends Publisher<? extends T>> mapper) {          this.source = source;          this.mapper = mapper;      }        @Override      public void subscribe(Subscriber<? super R> s) {          T value;            try {              value = source.call();                                    // (1)          } catch (Throwable ex) {              ExceptionHelper.throwIfFatal(ex);              EmptySubscription.error(s, ex);              return;          }            if (value == null) {              EmptySubscription.complete(s);              return;          }            Publisher<? extends R> p;            try {              p = mapper.apply(value);                                  // (2)          } catch (Throwable ex) {              ExceptionHelper.throwIfFatal(ex);              EmptySubscription.error(s, ex);              return;          }            if (p == null) {              EmptySubscription.error(s,                  new NullPointerException("The mapper returned null");              return;          }            if (p instanceof Callable) {                                  // (3)              R result;                try {                  result = ((Callable<R>)p).call();              } catch (Throwable ex) {                  ExceptionHelper.throwIfFatal(ex);                  EmptySubscription.error(s, ex);                  return;              }                if (result == null) {                  EmptySubscription.complete(s);                  return;              }                s.onSubscribe(new ScalarSubscription<>(s, result));                return;          }            p.subscribe(s);      }  } |

First (1), we extract the single value from the underlying **Callable** instance. If it is **null**, we complete the **Subscriber** immediately. Otherwise, we call the mapper that returns a **Publisher** (2). Since this publisher could be also a **Callable**, we do the extraction again (3) and either complete or set a backpressure-enabled **ScalarSubscription** on the **Subscriber**. Because **call()** can throw, we catch any exception, signal the fatal ones in some library-specific way and signal non-fatal exceptions to the **Subscriber** as well (plus setting its **Subscription** at the right time).

#### Caution with Callable

Since **Callable** is an established interface, one must be careful with implementors of **Publisher** and **Callable** where functionally, the callable means something different than a shortcut to a 0 or 1 element.  
  
My hope is that since RS is relatively new and only a few people have actually implemented operators with it, we can avoid any pitfalls related to this combined interface approach.

### Micro-fusion constructs

Unlike macro-fusion, micro-fusion requires a protocol switch between two subsequent operators; instead of using the standard RS method calls, some or all of them gets replaced by other method calls. This allows sharing internal structures or state between the two.  
  
In theory, in a pair of operators, the upstream operator can be the initiator and work with the internals of the downstream operator. In practice, so far, we implemented fusion the other way around: the downstream operator works with the internals of the upstream operator.  
  
However, going for a full custom interaction is not advised because that may lead to a complete custom implementation and duplication of a lot of code. (That being said, unfortunately, **ConditionalSubscriber** requires code duplication to avoid casting.)  
  
Currently, Rsc and Reactor can do two kinds of micro-fusion: conditional and queue-based. On a second dimension, we can think 3 kinds of operators:

* sources that support fusion (**range()**, **UnicastProcessor**)
* intermediate operators that may support fusion (**concatMap**, **observeOn**, **groupBy, window)**
  + front fusion (**concatMap**)
  + back fusion (**groupBy**)
  + transitive fusion (**map**, **filter**)
* consumers (**flatMap** inner, **zip**)

The third dimension appears with queue-based fusion where the source can be synchronous (i.e., **fromArray**) or asynchronous (**UnicastProcessor**).

#### Conditional micro-fusion

The conditional micro-fusion ability is indicated by an interface: **ConditionalSubscriber** extending **Subscriber** with one extra method:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3 | public interface ConditionalSubscriber<T> extends Subscriber<T> {      boolean onNextIf(T t);  } |

If a source or intermediate operator sees that its consumer is a **ConditionalSubscriber** it may call the **onNextIf** method. (By nature, this means a synchronous execution and response, thus conditional fusion is for synchronous cases only.)  
  
If the method returns **true**, the value has been consumed as usual. If the method returns **false**, it means the value was dropped and a new value can be sent immediately. This avoids a **request(1)** call for a replenishment in **filter** and other operators as well.  
  
*Sidenote: You may ask, why is this important? A call to* ***request()*** *usually ends up in an atomic CAS, costing 21-45 cylces for each dropped element.*

To work with **ConditionalSubscriber**s in source operators, you may have to first switch on the incoming **Subscriber**'s type and do a different implementation to avoid casting the downstream **Subscriber** all the time.

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Override  public void subscribe(Subscriber<? super Integer> s) {      if (s instanceof ConditionalSubscriber) {            s.onSubscribe(new RangeConditionalSubscription<>(              (ConditionalSubscriber<T>)s, start, count));        } else {          s.onSubscribe(new RangeSubscription<>(s, start, count);      }  } |

The implementation can then can use the **onNextIf** method during emissions. For example, the fast-path can be rewritten as follows:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | for (long i = start; i < (long)start + count; i++) {      if (cancelled) {          return;      }      s.onNextIf((int)i);  }  if (!cancelled) {      s.onComplete();  } |

You may think, why call **onNextIf** if we don't care about the return value? For composition reasons. Even though this path in **range()** doesn't need the return value, but if the downstream is also calling **onNextIf** further down, this can avoid a whole chain of unnecessary **request(1)** calls.  
  
The slow path is more interesting in this regard:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | long i = index;  long end = (long)start + count;  long r = requested;  long e = 0L;    while (i != end && e != r) {      if (cancelled) {         return;      }        if (s.onNextIf((int)i)) {          e++;      }      i++;  }    if (i == end) {      if (!cancelled) {          s.onComplete();      }      return;  }    if (e != 0L) {      index = i;      REQUESTED.addAndGet(this, REQUESTED, -e);  } |

In the while loop, if the **onNextIf** returns false, we don't increment the emission count which means the next integer value can come immediately. If a downstream consumer requests only 1 and then drops all values, the loop can exhaust the available integers and not call the atomic **addAndGet** even once.  
  
Since filter is one of the most common operators in a chain, one should be prepared to work with **ConditionalSubscriber** even if one doesn't interfere with the number of events flowing through. For example, **map()** and filter appear together and it is advised **map()** also supports conditional fusion by switching on the **Subscriber**'s type just like above and using a **ConditionalSubscriber**-based **Subscriber**:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | static final class MapConditionalSubscriber<T, R> implements ConditionalSubscriber<T> {      final ConditionalSubscriber<? super R> actual;        final Function<? super T, ? extends R> mapper;        boolean done;        Subscription s;        // ...        @Override      public boolean onNextIf(T t) {          if (done) {              return;          }            R v;            try {              v = mapper.apply(t);          } catch (Throwable ex) {              ExceptionHelper.throwIfFatal(ex);              s.cancel();              onError(ex);              return;          }            if (v == null) {              s.cancel();              onError(new NullPointerException("..."));              return;          }            return actual.onNextIf(v);      }        // ...  } |

The final case for the conditional micro-fusion is the "terminal" operator or consumer implementation. Luckily, usually doesn't have to provide two implementations, on **ConditionalSubscriber** and one **Subscriber**, but have them together. Those who can work with the **ConditionalSubscriber** part will do, others will just use the regular **Subscriber** methods:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | static final FilterSubscriber<T> implements ConditionalSubscriber<T> {      final Subscriber<? super T> actual;        final Predicate<? super T> predicate;        boolean done;        Subscription s;        // ...        @Override      public void onNext(T t) {            if (!onNextIf(t)) {             s.request(1);          }      }        @Override      public boolean onNextIf(T t) {          if (done) {              return;          }            boolean pass;            try {              pass = predicate.test(t);          } catch (Throwable ex) {              ExceptionHelper.throwIfFatal(ex);              s.cancel();              onError(ex);          }            if (pass) {              actual.onNext(t);              return true;          }          return false;      }        // ...  } |

In conclusion, conditional micro-fusion is a relatively simple but sometimes verbose way of avoiding **request(1)** calls and the resulting per-item overhead.

#### Queue-based micro-fusion

Believe me if I tell, this is the most complicated thing, so far, in the reactive landscape. Not because it requires complicated structures or algorithms, but for the implications towards operators and the combinatoric-explosion nature of what happens if op1 is followed by op2 and how they can or can't fuse.  
  
The queue-based micro-fusion is built upon the idea that many operators employ a queue to work out backpressure-related or asynchrony-related cases when notifying the downstream and happen to face their queue towards each other.  
  
For example, **UnicastProcessor** has a backend-queue that holds values until the downstream requests them whereas **concatMap** has a front-queue that holds the source values to be mapped into **Publisher**s. When subscribed, a value goes from one queue into the other, forming a dequeue-enqueue pair without anything functional between the two other than the atomics overhead of request management and wip-accounting.  
  
Clearly, if we could somehow use a single queue between the two and somehow decrease the atomics overhead via it, we'd have a much lower overhead in terms of computation and memory usage.

However, what if there is an operator between the two that does something with the values? What if the fusion shouldn't happen in this case?  
  
To solve this coordination problem, we can reuse the onSubscribe(Subscription) rail in RS and extend the protocol. Enter **QueueSubscription**.

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public interface QueueSubscription<T> extends Queue<T>, Subscription {        int NONE = 0;      int SYNC = 1;      int ASYNC = 2;      int ANY = SYNC | ASYNC;      int THREAD\_BOUNDARY = 4;        int requestFusion(int mode);        @Override      default boolean offer(T t) {          throw new UnsupportedOperationException();      }        // ...  } |

The **QueueSubscription** is a combination of **Queue** and **Subscription** interfaces, adding a new **requestFusion()** method, and other than keeping the following methods from the base interfaces, all other methods are defaulted to **UnsupportedOperationException** as we won't need them. (Java 7- note: yes you may have to manually do this for classes that can't extend a base class.):

* **void request(long n);**
* **void cancel()**
* **T poll()**
* **boolean isEmpty()**
* **void clear();**

(Some libraries may choose to implement **size()** as well, for diagnostic purposes.)

When a source supports queue-based fusion, it can send a **QueueSubscription** implementation through **onSubscribe**. Those who can deal with it can act on it, the rest will simply see it as a regular **Subscription**.  
  
The idea is that those who can deal with it can use it as a **Queue** instead of instantiating their own, saving on allocation and overhead at the same time. In addition, a source such as **range()** can itself pretend to be a queue, returning the next value through **poll()** or null if no more integers remain.  
  
Since there are cases where fusion can't or should not happen, we need to perform a protocol switch during the subscription phase of a flow. This switch can be requested via the **requestFusion()** method, that takes and returns the constants from the interface.  
  
*(Sidenote: I know enums would be more readable, but* ***EnumSet*** *has a nice additional overhead you know...)*  
  
As an input, it can take:

* **SYNC** - indicates the consumer wants to work with a synchronous upstream, with often known length
* **ASYNC** - indicates the consumer wants to work with an asynchronous upstream with often unknown length and emission timing
* **ANY** - indicates a consumer can work with both **SYNC** and **ASYNC** upstream
* **(SYNC, ASYNC) | THREAD\_BOUNDARY** - indicates that the consumer goes over a thread boundary and **poll()** happens on some other thread.

It can return:

* **NONE** - fusion can't happen/rejected
* **SYNC** - synchronous fusion mode activated
* **ASYNC** - asynchronous fusion mode activated

If the upstream is unable to work in the requested mode or is sensitive to thread-boundary effects, it can return **NONE**. In this case, the flow behaves just like the regular, non-fused RS stream would. (Note that conditional fusion is still may be an option.)

Because fusion is optional, a successfully negotiated mode requires different mode of execution in either or both parties. In addition, this mode switch has to happen before any events fly through the chain, therefore, **onSubscribe** is an ideal place for it due to the underlying RS protocol spec.

Both **SYNC** and **ASYNC** modes have extra rules implementors must adhere.

In **SYNC** mode, consumers should never call **request()** and producers should never return **null** from **poll()** unless they mean completion. Since the only interaction between the two are through **poll()** and **isEmpty()**, sources have no opportunity to call **onError** but must throw a runtime exception from these two methods. On the other side, consumers now have to wrap these methods into try-catches and handle/unwrap exceptions there.

In **ASYNC** mode, the producer enqueues events in its own queue and has to signal the availability to the consumer. The best way for this is through **onNext**. One can either signal the value itself or null - the only place where you can do this. On the consumer side, the **ASYNC** mode **onNext** now has meaningless value and should be ignored. The other methods, **onError**, **onComplete**, **request** and **cancel** should be used as in regular RS cases. In this mode, **poll()** can return **null** indicating a temporary lack of values; the termination will be indicated by **onError** and **onComplete** as usual.

**Implementing fusion-enabled sources**

Now let's see the API in action. First, let's make **range()** fusion enabled:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | static final class RangeSubscription extends QueueSubscription<Integer> {        // ... the Subscription part is the same        @Override      public Integer poll() {          long i = index;          if (i == (long)start + count) {              return null;          }          index = i + 1;          return (int)i;      }        @Override      public boolean isEmpty() {          return index == (long)start + count;      }        @Override      public void clear() {          index = (long)start + count;      }        @Override      public int requestFusion(int mode) {          return SYNC;      }  } |

No sign of request accounting whatsoever because **range()** works in **synchronous pull mode**; consumer does backpressure by calling **poll()** when it needs a new value.

**UnicastProcessor** (which is somewhat like **onBackpressureBuffer()**) can support fusion in **ASYNC** mode specifically:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | public final class UnicastProcessor<T> implements Processor<T, T>, QueueSubscription<T> {        volatile Subscriber<? super T> actual;        final Queue<T> queue;        int mode;        // ...        @Override      public void onNext(T t) {          Subscriber<? super T> a = actual;          if (mode == ASYNC && a != null) {              a.onNext(null);          } else {              queue.offer(t);              drain();          }      }        @Override      public int requestFusion(int m) {          if ((m & ASYNC) != 0) {              mode = ASYNC;              return ASYNC;          }          return NONE;      }        @Override      public T poll() {          return queue.poll();      }        @Override      public boolean isEmpty() {          return queue.isEmpty();      }        @Override      public void clear() {          queue.clear();      }        @Override      public void subscribe(Subscriber<? super T> s) {          if (ONCE.compareAndSet(this, 0, 1)) {              s.onSubscribe(this);              actual = s;              if (cancelled) {                  actual = null;              } else {                  if (mode != NONE) {                      if (done) {                          if (error != null) {                              s.onError(error);                          } else {                              s.onComplete();                          }                      } else {                          s.onNext(null);                      }                  } else {                      drain();                  }              }          } else {              EmptySubscription.error(s, new IllegalStateException("..."));          }      }  } |

The fusion mode requires the following behavior changes:

* **onNext** has to call **actual.onNext** instead of **drain()**,
* **requestFusion** has to see if the downstream actually wants **ASYNC** fusion,
* the queue methods have to be delegated to the instance **queue**,
* the **subscribe()** has to call **actual.onNext** instead of **drain()** as well.

Doesn't look too complicated, does it? At this point, you can check your understanding of supporting fusion through an exercise: can **UnicastProcessor** support **SYNC** fusion and if so, when and how; if not, why not?  
  
  
**Implementing fusion-enabled intermediate operators**   
  
In practice, usually there are some intermediate operators between a fuseable source and a fusion-enabled consumer. Unfortunately, this can break the fusion (and thus reverting to the classical RS) mode or worse, the data may skip the intermediate operator altogether, causing all sorts of failures.  
  
The latter manifests itself when an operator forwards the Subscription it received via its onSubscribe method. Now imagine if map() does this; what would be the output of the following sequence:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1 | range(0, 10).map(v -> v + 1).concatMap(v -> just(v)).subscribe(System.out::println); |

In a classical flow, you'd get values 1 through 10 printed to the console. If both range() and concatMap() do fusion but map() forwards its Subscription, the surprising output is 0 through 9! This can affect any operator.  
  
The solution is to require all operators that don't want to participate in fusion to never forward the upstream's subscriber verbatim. A possible manifestation of this rule is to implement Subscription on yourself:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | static final class MapSubscriber<T, R> implements Subscriber<T>, Subscription {      // ...        @Override      public void onSubscribe(Subscription s) {          this.s = s;            actual.onSubscribe(this);      }        @Override      public void request(long n) {          s.request(n);      }        @Override      public void cancel() {          s.cancel();      }        // ...  } |

In practice, many operators that either manipulate requests or cancellation does this so the indirection is an acceptable trade-off for the benefit of a lower overhead dataflow in general.  
  
This rule, unfortunately affects cross-library behavior. Even though other libraries may not speak the same fusion protocol, they could end up forwarding **Subscription**s, thus if you go into and out of some other library, the same problem may appear again. Generally, libraries supposed to have a method **hide()** or **asObservable()** to hide the identity of a source as well as preventing the propagation of unwanted internal features.  
  
Luckily, **map()** can participate in the fusion: it only has to be fuseable itself, mediate the **requestFusion** between its upstream and downstream, plus place itself at the exit point: **poll()**.

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | static final class MapSubscriber<T, R> implements Subscriber<T>, QueueSubscription<R> {      final Subscriber<? super R> actual;        final Function<? super T, ? extends R> mapper;        QueueSubscription<T> qs;        Subscription s;        int mode;        // ...        @Override      public void onSubscribe(Subscription s) {          this.s = s;          if (s instanceof QueueSubscription) {              qs = (QueueSubscription<T>)s;          }            actual.onSubscribe(this);      }        @Override      public void onNext(T t) {          if (mode == NONE) {                // error handling omitted for brevity                actual.onNext(mapper.apply(t));            } else {              actual.onNext(null);          }      }        @Override      public int requestFusion(int m) {          if (qs == null || (m & THREAD\_BOUNDARY) != 0) {              return NONE;          }          int u = qs.requestFusion(m);          mode = u;          return u;      }        @Override      public R poll() {          T t = qs.poll();          if (t == null) {              return null;          }          return mapper.apply(t);      }        @Override      public boolean isEmpty() {          return qs.isEmpty();      }        @Override      public void clear() {          qs.clear();      }  } |

The operator **map()** can implement **QueueSubscription** itself and have a field for the potential upstream's **QueueSubscription** as well. In **requestFusion**, if the upstream does support fusion and the downstream isn't a boundary, the request is forwarded to upstream; rejected otherwise.  
  
Now **poll()** can't just forward to the upstream because the types are different. Here comes the mapper function that is applied to the upstream's value. Note that **null** indicates termination or temporary lack of values and should not be mapped.  
  
The main reason **THREAD\_BOUNDARY** was introduced as a flag is due to **map()**, or in a more broader sense: the restriction on where user-supplied computations happen. In fusion mode, the execution of the mapper function happens on the exit side of the queue, which could be in some other thread. Now imagine you have a heavy computation in map which would run off the main thread before reaching an **observeOn**. When unfused, the result of the computation would be queued up in **observeOn**, then dequeued on the target thread (let's say the main thread). However, if fusion is allowed, the target thread is doing the **poll()** and now the heavy calculation runs on the main thread.  
  
The operator **filter()** can be implemented in a similar fashion, but our old **request(1)** comes back unfortunately:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | static final class FilterSubscriber<T> implements Subscriber<T>, QueueSubscription<T> {      // ...        @Override      public T poll() {          for (;;) {              T v = qs.poll();                if (v == null || cancelled) {                  return null;              }                if (predicate.test(v)) {                  return v;              }                if (mode == ASYNC) {                  qs.request(1);              }          }      }        @Override      public boolean isEmpty() {          return qs.isEmpty();      }        // ...  } |

Since **filter()** drops values, we need to loop in **poll()** until the **predicate** matches or no more upstream values are available for some reason. If the **predicate** doesn't match, we have to replenish our **ASYNC** source (remember, you are not supposed to call **request()** in sync mode!).  
  
**Implementing fusion-enabled consumers**  
  
Generally, operator fusion is not very useful (or really happens) with end-subscribers, such as your favorite **Subscriber** subclass or with **subscribe(System.out::println)**.  
  
The consumers I'm talking about can be considered intermediate operators as well, but since all operators are basically custom **Subscriber**s that are subscribed to the upstream, they are consumers as well.  
  
As I mentioned, many operators feature some internal queue on their front side (e.g., **concatMap**, **observeOn**) or when they consume some inner Publisher (i.e., **flatMap,** **zip**). These are the primary consumers and drivers of the fusion lifecycle.  
  
Now that [we are familiar](http://akarnokd.blogspot.hu/2016/03/subscribeon-and-observeon.html) with how **observeOn** is implemented, let's see how can we enable fusion with it:

[?](https://akarnokd.blogspot.nl/2016/04/operator-fusion-part-2-final.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | static final class ObserveOnSubscriber<T> implements Subscriber<T>, Subscription {        Queue<T> queue;        int mode;        Subscription s;        // ...        @Override      public void onSubscribe(Subscription s) {           this.s = s;             if (s instanceof QueueSubscription) {               QueueSubscription<T> qs = (QueueSubscription<T>)s;                 int m = qs.requestFusion(QueueSubscription.ANY                    | QueueSubscription.THREAD\_BOUNDARY);                 if (m == QueueSubscription.SYNC) {                   q = qs;                   mode = m;                   done = true;                     actual.onSubscribe(this);                     return;               }                 if (m == QueueSubscription.ASYNC) {                   q = qs;                   mode = m;                     actual.onSubscribe(this);                     s.request(prefetch);                     return;               }           }             queue = new SpscArrayQueue<>(prefetch);             actual.onSubscribe(this);             s.request(prefetch);      }        @Override      public void onNext(T t) {          if (mode == QueueSubscription.NONE) {              queue.offer(t);          }            drain();      }        void drain() {            // ...                if (mode != QueueSubscription.SYNC) { |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | request(p);              }            // ...        }        // ...    } |

Enabling fusion has two implications: 1) **queue** can no longer be final but has to be created in **onSubscribe**, 2) **onNext** should not offer if fusion is enabled.  
  
The fusion mode is requested in the **onSubscribe** after identifying the upstream as **QueueSubscription**. Since the algorithm inside **drain()** only sees the **Queue** interface and doesn't particularly care when values are available in the queue, we request the **ANY** mode from upstream in addition to indicating the consumer is also a **THREAD\_BOUNDARY**. This should prevent the **poll()** side to change the location of some user-defined function unexpectedly.  
  
If **SYNC** mode is granted, we assign the **QueueSubscription** to our **queue** and call **onSubscribe** on the downstream **Subscriber**. In this mode, the prefetch amount is not requested in accordance with the synchronous fusion protocol. The big win in **SYNC** mode is the fact that if **poll()** returns **null**, that is an indication of termination. We already exploit this in the standard queue-drain algorithm: if the **done** flag is set and the queue reports **null**/empty, we have completed. Note however, that we have to adjust the drain algorithm a bit because we can't call **request** in **SYNC** mode anymore.  
  
If **ASYNC** mode is granted, we store the **queue** again, but can't set the **done** flag as we don't know when the upstream finishes - **poll()** returning **null** is just the indication of unavailability of values at the time. In addition, once the downstream **Subscriber** is notified, we still have to signal a **prefetch**-request to upstream, so it can trigger its own sources even further up.  
  
Note that once **requestFusion** returns **SYNC** or **ASYNC**, there is no going back (you may try to call **requestFusion()** again which may change the mode, but that's undefined behavior at the moment; it may be forbidden entirely in the future), definitely not after elements have been delivered already in any mode.

#### General warnings around micro-fusion

In my experience, some of my colleagues tend to become enthusiastic about micro-fusion; they want to apply it everywhere. Whenever an operator has any queue, they see fusion happening.  
  
I must warn against such relentlessness because fusion has some requirements, implications and generally subject to cost-benefit trade-offs:

* If an operator is a thread boundary, my current understanding is that you can't fuse both its front and back side at the same time.
* Fusion can shift computations in time and sometimes in location (even without an explicit boundary).
* The fact an operator has a queue doesn't mean it can be exposed/replaced. A good example of this is **combineLatest**: my current understanding is that the post-processing of the queue elements makes this infeasible for back side fusion. Another example is **flatMap** where I'm not convinced the collector logic can be integrated into a **poll()**/**isEmpty()** back-side fusion.
* Some sources, such as 0 or 1 are likely not worth it and are better off with macro-fusions.
* Fusion is an extra behavior which also can be buggy or in fact, hide a bug on the regular path (i.e., groupBy) and requires extra care. In addition, it increases the test method count because now you have to test with and without fusion (see **hide()**).

To cheer you up, there is a great counter-example operator that supports full fusion: front and back side at the same time: **flattenIterable**, or as you may know it, **concatMapIterable**/**flatMapIterable**.

### Conclusion

In this post, I've detailed the structures and protocols of operator fusion and shown some examples how it can be utilized in source, intermediate and terminal operators.  
  
Since operator fusion is an active research area, I can't say these are all that can happen and we are eager to hear about interesting chains of operators where fusion can happen, or in contrast, were fusion should not happen. See the [Rsc repository](https://github.com/reactor/reactive-streams-commons) for examples of all kinds of fusions.  
  
In addition, I hope these fusion protocols will be standardized and be part of **Reactive-Streams 2.0**, allowing a full, cross-library efficient operation that maintains fusion as long as possible.  
  
My next topic will be to finish up the series about **ConnectableObservable**s.

### [Writing a custom reactive base type](https://akarnokd.blogspot.nl/2016/03/writing-custom-reactive-base-type.html)

### Introduction

From time to time, the question or request comes up that one would really like to have his/her own reactive type. Even though RxJava's **Observable** has plenty of methods and extension points via **lift()**, **extend()** and **compose()**, one feels the **Observable** should have the operator **xyz()** or in some chains, the chain shouldn't allow calling **uvw()**.  
  
The first case, namely adding a new custom method without going through the project as a contribution, is as old as the reactive programming on the JVM. When I first ported Rx.NET to Java, I had to face the same problem because .NET had the very convenient extension method support already back in 2010. Java doesn't have this and the idea has been rejected in the version 8 development era in the favor of default methods with the "justification" that such extension methods can't be overridden. Yes they can't but they can be replaced by another method from another class.  
  
The second case, hiding or removing operators, comes up with custom **Observable**s where certain operations don't make sense. For example, given a **ParallelObservable** that splits the input sequence into parallel processing pipelines internally, it makes sense to **map()** or **filter()** in parallel, but it doesn't make sense to use **take()** or **skip()**.

### Wrapping

Both cases can be solved by writing a custom type and just wrap the **Observable** into it.

public final class MyObservable<T> {

    private Observable<T> actual;

    public MyObservable<T>(Observable<T> actual) {

        this.actual = actual;

    }

}

Now we can add operators of our liking:

    // ...

    public static <T> MyObservable<T> create(Observable<T> o) {

        return new MyObservable<T>(o);

    }

    public static <T> MyObservable<T> just(T value) {

        return create(Observable.just(value));

    }

    public final MyObservable<T> goAsync() {

        return create(actual.subscribeOn(Schedulers.io())

            .observeOn(AndroidSchedulers.mainThread()));

    }

    public final <R> MyObservable<R> map(Func1<T, R> mapper) {

        return create(actual.map(mapper));

    }

public final void subscribe(Subscriber<? super T> subscriber) {

actual.subscribe(subscriber);

}

    // ...

As seen here, we achieved both goals: get rid of the unnecessary operators and introduce our own operator while staying within our custom type.  
  
If you look at the source code of RxJava, you see the same pattern where the actual object is just the **OnSubscribe** / **Publisher** type and the **Observable** enriches them with all sorts of operators.

### Interoperation

The **MyObservable** looks adequate, but eventually, one has to interoperate with the regular **Observable** or somebody else's **YourObservable**. Because these are distinct types, we need a common type they can communicate with each other. Naturally, everybody can implement a **toObservable()** and return an **Observable** view, but that's yet another inconvenience of calling the method. Instead, every **MyObservable** and **YourObservable** can extend a base class or implement an interface with the minimal set of operations that each requires.  
  
In RxJava 1.x, the obvious choice, **Observable**, isn't too good, because its methods are final and leak into **MyObservable** and the worst, they all return **Observable** instead of **MyObservable**! Unfortunately, 1.x can't help in this regard due to binary compatibility reasons.  
  
Lucky thing is that in 2.x, **Observable** (**Flowable**) isn't really the root of the reactive type but **Publisher**. Every observable is a **Publisher** and many operators take **Publisher** as parameter instead of **Observable**. This has the benefit of working with other **Publisher**-based types out of box. The reason this can work is that for the **Observable** chain to work, the operators only need a single method to be available from their sources: **subscribe(Subscriber<? super T> s);**  
  
Therefore, if we target 2.x, the **MyObservable** should implement **Publisher** and thus immediately available as source to operators of any decent reactive library:

public class MyObservable<T> implements Publisher<T> {

    // ...

    @Override

    public void subscribe(Subscriber<? super T> subscriber) {

        actual.subscribe(subscriber);

    }

    // ...

}

### Extension

Given this **MyObservable**, one would eventually want to have other custom reactive type for different use cases, but that will become tedious as well due to the need for duplicating operators all over. Naturally, one thinks about using **MyObservable** as the base class for **TheirObservable** and adding new operators there, but that suffers from the same problem as the **Observable** -> **MyObservable** would: operators return the wrong type.  
  
I believe the Java 8 Streams API suffered from a similar problem and if you look at the signature, **Stream extends BaseStream<T, Stream<T>>** and **BaseStream<T, S extends BaseStream<T, S>>**. Quite odd that some supertype has a type parameter for the subtype. The reason for this is to capture the subtype in the type signature of the methods, thus if you have **MyStream**, all stream methods' type signature now has **MyStream** as a return type.  
  
We can achieve a similar structure by declaring MyObservable as follows:

    public class MyObservable<T, S extends MyObservable<T, S>> implements Publisher<T> {

        final Publisher<? extends T> actual;

        public MyObservable(Publisher<? extends T> actual) {

            this.actual = actual;

        }

        @SuppressWarnings("unchecked")

        public <R, U extends MyObservable<R, U>> U wrap(Publisher<? extends R> my) {

            return (U)new MyObservable<R, S>(my);

        }

        public final <R, U extends MyObservable<R, U>> U map(Function<? super T, ? extends R> mapper) {

            return wrap(Flowable.fromPublisher(actual).map(mapper));

        }

        @Override

        public void subscribe(Subscriber<? super T> s) {

            actual.subscribe(s);

        }

    }

Quite a set of generic type mangling. We specify a **wrap()** method that turns an arbitrary **Publisher** into **MyObservable** and we call it from **map()** to ensure the result type is ours. Descendants of **MyObservable** will then override wrap to provide their own type:

    public class TheirObservable<T> extends MyObservable<T, TheirObservable<T>> {

        public TheirObservable(Publisher<? extends T> actual) {

            super(actual);

        }

        @SuppressWarnings("unchecked")

        @Override

        public <R, U extends MyObservable<R, U>> U wrap(Publisher<? extends R> my) {

            return (U) new TheirObservable<R>(my);

        }

    }

Let's try it:

    public static void main(String[] args) {

        TheirObservable<Integer> their = new TheirObservable<>(Flowable.just(1));

        TheirObservable<String> out = their.map(v -> v.toString());

        Flowable.fromPublisher(out).subscribe(System.out::println);

    }

It works as expected; no compilation errors and it prints the number 1 to the console.  
  
Now let's add a **take()** operator to **TheirObservable**:

        @SuppressWarnings({ "rawtypes", "unchecked" })

        public <U extends AllObservable<T>> U take(long n) {

            Flowable<T> p = Flowable.fromPublisher(actual);

            Flowable<T> u = p.take(1);

            return (U)(AllObservable)wrap(u);

        }

The method signatures get more complicated and the type system starts to fight back; one needs raw types and casts to make things appear the expected type. In addition, if one writes **their.map(v -> v.toString()).take(1);** the compiler won't find **take()**. The reason for it is that map returns something of **MyObservable** which something was defined by the assignment to **TheirObservable**. To make the types work out, we have to split the fluent calls into individual steps:

        TheirObservable<Integer> their2 = new TheirObservable<>(Flowable.just(1));

        TheirObservable<String> step1 = their2.map(v -> v.toString());

        TheirObservable<String> step2 = step1.take(1);

        Flowable.fromPublisher(step2).subscribe(System.out::println);

Finally, lets extend **TheirObservable** further into **AllObservable** and let's add the **filter()** method:

public static class AllObservable<T> extends TheirObservable<T> {

public AllObservable(Publisher<? extends T> actual) {

super(actual);

}

@Override

<R, U extends MyObservable<R, U>> U wrap(Publisher<? extends R> my) {

return (U)new AllObservable<R>(my);

}

@SuppressWarnings({ "rawtypes", "unchecked" })

public <U extends AllObservable<T>> U filter(Predicate<? super T> predicate) {

Flowable<T> p = Flowable.fromPublisher(actual);

Flowable<T> u = p.filter(predicate);

return (U)(AllObservable)wrap(u);

}

}

then use it:

AllObservable<Integer> all = new AllObservable<>(Flowable.just(1));

AllObservable<String> step1 = all.map(v -> v.toString());

AllObservable<String> step2 = step1.take(1);

AllObservable<String> step3 = step2.filter(v -> true);

Flowable.fromPublisher(step3).subscribe(System.out::println);

Unfortunately, this doesn't compile because map() doesn't return **AllObservable**, namely, **AllObservable** is not **MyObservable<String, U extends MyObservable<String, U>>**. Changing **step1**'s type to **TheirObservable<String>** resolves the compilation issue. However, if one would then swap **filter()** and **take()**, **step1** no longer is an **AllObservable** and **filter()** is no longer available.

### Conclusion

Can we fix the situation with **AllObservable**? I don't know; this is where my understanding of Java's type system and type inference ends.  
  
Will RxJava 2.x have such structure then? If it were up to me then no. To support this style, we'd need wrapping all the time despite I want to get rid of all **lift()** and **create()** use and the type signatures of classes and methods end up way more complicated than before.  
  
Therefore, if one wants to go down this path, the example shows above that RxJava's API doesn't have to change and can be wrapped to do the work while one specifies their surface API at will. It is a good example for "composition over inheritance".

### [SubscribeOn and ObserveOn](https://akarnokd.blogspot.nl/2016/03/subscribeon-and-observeon.html)

### Introduction

One of the most confused operator pair of the reactive ecosystem is the **subscribeOn** and **observeOn** operators. The source of confusion may be rooted in a few causes:

* they sound alike,
* they sometimes show similar behavior when looked at from downstream and
* they are duals in some sense.

It appears the name-confusion isn't local to RxJava. Project Reactor faces a similar issue with their **publishOn** and **dispatchOn** operators. Apparently, it doesn't matter what they are called and people will confuse them anyhow.

When I started learning about Rx.NET back in 2010, I never experienced this confusion; **subscribeOn** affects s**ubscribe()** and **observeOn** affects **onXXX()**.

*(Remark: I've searched Channel 9 for the early videos but couldn't really find the talk where they build up these operators just like I'm about to do. The closest thing was* [*this*](https://channel9.msdn.com/Blogs/J.Van.Gogh/Controlling-concurrency-in-Rx)*.)*

My "thesis" is that the confusion may be resolved by walking through how one can implement these operators and thus showing the internal method-call flow.

### SubscribeOn

The purpose of **subscribeOn()** is to make sure side-effects from calling **subscribe()** happens on some other thread. However, almost no standard RxJava source does side-effects on its own; you can have side-effects with custom **Observable**s, wrapped subscription-actions via **create()** or as of lately, the with the **SyncOnSubscribe** and **fromCallable()** APIs.

Why would one move the side-effects? The main use cases are doing network calls or database access on the current thread or anything that involves blocking wait. Holding off a Tomcat worker thread hasn't been much of a programming problem (that doesn't mean we can't improve the stack with reactive) but holding off the Event Dispatch Thread in a Swing application or the Main thread in an Android application has adverse effect on the user experience.

*(Sidenote: it's a funny thing that blocking the EDT is basically a convenience backpressure strategy in the GUI world to prevent the user from changing the application state while some activity was happening.)*

Therefore, if the source does something immediately when a child subscribes, we'd want it to happen somewhere off the precious current thread. Naturally, we could just submit the whole sequence and the call to **subscribe()** to an **ExecutorService**, but then we'd be faced with the problem of cancellation being separate from the **Subscriber**. As more and more (complex) sequences requiring this asynchronous subscription behavior, the inconvenient it becomes to manage all those in this manner.

Luckily, we can include this behavior into an operator we call **subscribeOn()**.

For simplicity, let's build this operator on a much simpler reactive base type: the original **IObservable** from Rx.NET:

@FunctionalInterface

interface IObservable<T> {

    IDisposable subscribe(IObserver<T> observer);

}

@FunctionalInterface

interface IDisposable {

    void dispose();

}

interface IObserver<T> {

  void onNext(T t);

    void onError(Throwable e);

    void onCompleted();

}

Don't worry about synchronous cancellation and backpressure for now.

Let's assume we have a source which just sleeps for 10 seconds:

IObservable<Object> sleeper = o -> {

    try {

        Thread.sleep(1000);

        o.onCompleted();

    } catch (InterruptedException ex) {

        o.onError(ex);

    }

};

which will obviously go to sleep if we call **sleeper.subscribe(new IObserver ... );** Let's now create an operator that moves this sleep to some other thread:

ExecutorService exec = Executors.newSingleThreadedExecutor();

IObservable<Object> subscribeOn = o -> {

    Future<?> f = exec.submit(() -> sleeper.subscribe(o));

    return () -> f.cancel(true);

}

The **subscribeOn** instance will submit the action that subscribes to the actual **IObservable** to the executor and returns a disposable that will cancel the resulting Future from the submission.

Of course, one would instead have this in some static method or on a wrapper around the **IObservable** (as Java doesn't support extension methods):

public static <T> IObservable<T> subscribeOn(IObservable<T> source,

ExecutorService executor);

public Observable<T> subscribeOn(Scheduler scheduler);

Two of the common questions regarding **subscribeOn** are what happens when one applies it twice (directly or some regular operators in between) and why can't one change the original thread with a second **subscribeOn**. I hope the answer becomes apparent from the simplified structure above. Let's apply the operator a second time:

ExecutorService exec2 = Executors.newSingleThreadedExecutor();

IObservable<Object> subscribeOn2 = o -> {

Future<?> f2 = exec2.submit(() -> subscribeOn.subscribe(o));

return () -> f2.cancel(true);

};

Now let's expand **subscribeOn.subscribe()** in place:

IObservable<Object> subscribeOn2 = o -> {

Future<?> f2 = exec2.submit(() -> {

Future<?> f = exec.submit(() -> {

sleeper.subscribe(o);

});

});

};

We can simply read this from top to bottom. When **o** arrives, a task is scheduled on **exec2** which when executes, another task is scheduled on **exec** which when executes subscribes to **sleeper** with the original **o**. Becasue the **subscribeOn2** was the last, it gets executed first and no matter where it runs the task, it gets rescheduled by **subscribeOn** anyway on its thread. Therefore, that **subscribeOn()** operator's thread will matter which is closest to the source and one can't use another **subscribeOn()** application to change this. This is why APIs built on top of Rx either should not pre-apply **subscribeOn()** when they return an **Observable** or give the option to specify a scheduler.

Unfortunately, the **subscribeOn** operator above doesn't handle unsubscription properly: the result of the **sleeper.subscribe()** is not wired up to that external **IDisposable** instance and thus won't dispose the "real" subscription. Of course, this can be resolved by having a composite **IDisposable** and adding all relevant resources to it. In RxJava 1, however, we don't need this kind of juggling and the operator can be written with less work:

Observable.create(subscriber -> {

Worker worker = scheduler.createWorker();

    subscriber.add(worker);

    worker.schedule(

        () -> source.unsafeSubscribe(Schedulers.wrap(subscriber))

    )

});

This makes sure the **unsubscribe()** call on the subscriber will affect the **schedule()** and whatever resources the upstream source would use. We can use **unsafeSubscribe()** to avoid the unnecessary wrapping into a **SafeSubscriber** but we have to wrap the subscriber anyway because both **subscribe()** and **unsafeSubscribe()** call **onStart()** on the incoming **Subscriber**, which has already been called by the outer **Observable**. This avoids repeating any effects inside the user's **Subscriber.onStart()** method.  
  
The structure above composes backpressure as well, but we are not done.  
  
Before RxJava got backpressure, the **subscribeOn()** implementation above made sure that an otherwise synchronous source would emit all of its events on the same thread:

Observable.create(s -> {

for (int i = 0; i < 1000; i++) {

if (s.isUnsubscribed()) return;

s.onNext(i);

}

if (s.isUnsubscribed()) return;

s.onCompleted();

});

Users started to implicitly rely on this property. Backpressure breaks this property because usually the thread that calls **request()** will end up running the fragment of the loop above (see **range()**), causing potential thread-hopping. Therefore, to keep the same property, calls to **request()** has to go to the very same **Worker** that did the original subscription.  
  
The actual operator thus is more involved:

subscriber -> {

Worker worker = scheduler.createWorker();

subscriber.add(worker);

worker.schedule(() -> {

Subscriber<T> s = new Subscriber<T>(subscriber) {

@Override

public void onNext(T v) {

subscriber.onNext(v);

}

@Override

public void onError(Throwable e) {

subscriber.onError(e);

}

@Override

public void onCompleted() {

subscriber.onCompleted();

}

@Override

public void setProducer(Producer p) {

subscriber.setProducer(n -> {

worker.schedule(() -> p.request(n));

});

}

};

source.unsafeSubscribe(s);

});

}

Other than forwarding the **onXXX()** methods to the child subscriber, we set a custom producer on the child where the **request()** method schedules an action that calls the original producer with the same amount on the scheduler, ensuring that if there is an emission tied to the request, that happens on the same thread every time.  
  
This can be optimized a bit by capturing the current thread in the outer **schedule()** action, comparing it to the caller thread in the custom **Producer** and then calling **p.request(n)** directly instead of scheduling it:

Thread current = Thread.currentThread();

// ...

subscriber.setProducer(n -> {

if (Thread.currentThread() == current) {

p.request(n);

} else {

worker.schedule(() -> p.request(n));

}

});

### ObserveOn

The purpose of **observeOn** is to make sure values coming from any thread are received or observed on the proper thread. RxJava is by default synchronous, which technically means that **onXXX()** methods are called in sequence on the same thread:

for (int i = 0; i < 1000; i++) {

MapSubscriber.onNext(i) {

FilterSubscriber.onNext(i) {

TakeSubscriber.onNext(i) {

MySubscriber.onNext(i);

}

}

}

}

There are several use cases for moving this **onNext()** call (and any subsequent calls chained after) to another thread. For example, generating the input to a **map()** operation is cheap but the calculation itself is expensive and would hold off the GUI thread. Another example is when there is a background activity (database, network or the previous heavy computation), the results should be presented on the GUI and that requires the programmer to only interact with the GUI framework on the specific thread.  
  
In concept, **observeOn** works by scheduling a task for each **onXXX()** calls from the source on a specific scheduler where the original parameter value is handed to the downstream's **onXXX()** methods:

ExecutorService exec = Executors.newSingleThreadedExecutor();

IObservable<T> observeOn = o -> {

source.subscribe(new Observer<T>() {

@Override

public void onNext(T t) {

exec.submit(() -> o.onNext(t));

}

@Override

public void onError(Throwable e) {

exec.submit(() -> o.onError(e));

}

@Override

public void onCompleted() {

exec.submit(() -> o.onCompleted());

}

});

};

This pattern only works if the executor is single threaded or otherwise ensures FIFO behavior and doesn't execute multiple tasks from the same "client" at the same time.  
  
Unsubscription here is more complicated, because one has to keep track all the pending tasks, remove them when they have finished and make sure every pending task can be mass-cancelled.  
  
I believe the Rx.NET has some complicated machinery for this, but luckily, RxJava has a simple solution in the form of the **Scheduler.Worker**, taking care of all the required unsubscription behavior:

Observable.create(subscriber -> {

Worker worker = scheduler.createWorker();

subscriber.add(worker);

source.unsafeSubscribe(new Subscriber<T>(subscriber) {

@Override

public void onNext(T t) {

worker.schedule(() -> subscriber.onNext(t));

}

@Override

public void onError(Throwable e) {

worker.schedule(() -> subscriber.onError(e));

}

@Override

public void onCompleted() {

worker.schedule(() -> subscriber.onCompleted());

}

});

});  
  
Now if we compare **subscribeOn** and **observeOn**, one can see that **subscribeOn** schedules the entire **source.subscribe(...)** part whereas **observeOn** schedules the individual **subscriber.onXXX()** calls onto another thread.  
  
You can now see if **observeOn** is applied twice, that inner scheduled task expands to another lever of scheduling:

worker.schedule(() -> worker2.schedule(() -> subscriber.onNext(t)));

thus it overrides the emission thread in the chain, therefore, functionally, the closest **observeOn** to the consumer will win. From the expanded call above, you can see that **worker** is now wasted as a resource while providing no functional value to the sequence.  
  
The **observeOn** with the given structure has a drawback. If the source is some trivial **Observable** such as **range(0, 1M);** that will emit all of its values and suddenly, we have a large amount of pending task in the underlying threadpool of the scheduler. This can overwhelm the downstream consumer and also consumes lot of memory.  
  
Backpressure was introduced mostly to handle such cases, preventing internal buffer bloat and unbounded memory usage due to an asynchronous boundary. Consumers specifying the items they can consume via **request()** makes sure the producer side will only emit that many elements **onNext()**. Once the consumer is ready, it will issue another **request()**. The **observeOn()** above, with its **new Subscriber<T>(subscriber)** wrapping, already composes backpressure and relays the **request()** calls to the upstream source. However, this doesn't prevent the consumer from requesting everything via **Long.MAX\_VALUE** and now we have the same bloat problem again.  
  
Unfortunately, RxJava discovered the backpressure problem too late and the mandatory requesting would have required a lot of user code changes. Instead, backpressure was introduced as an optional behavior and made the resposibility of the operators such as **observeOn** to handle it while maintaining transparency with bounded **Subscriber**s and unbounded **Observer**s alike.  
  
The way it can be handled is via a queue, request tracking for the child **Subscriber**, fixed request amount towards the source and a queue-drain loop.

Observable.create(subscriber -> {

Worker worker = scheduler.createWorker();

subscriber.add(worker);

source.unsafeSubscribe(new Subscriber<T>(subscriber) {

final Queue<T> queue = new SpscAtomicArrayQueue<T>(128);

final AtomicLong requested = new AtomicLong();

final AtomicInteger wip = new AtomicInteger();

Producer p;

volatile boolean done;

Throwable error;

@Override

public void onNext(T t) {

queue.offer(t);

trySchedule();

}

@Override

public void onError(Throwable e) {

error = e;

done = true;

trySchedule();

}

@Override

public void onCompleted() {

done = true;

trySchedule();

}

@Override

public void setProducer(Producer p) {

this.p = p;

subscriber.setProducer(n -> {

BackpressureUtils.addAndGetRequested(requested, n);

trySchedule();

});

p.request(128);

}

void trySchedule() {

if (wip.getAndIncrement() == 0) {

worker.schedule(this::drain);

}

}

void drain() {

int missed = 1;

for (;;) {

long r = requested.get();

long e = 0L;

while (e != r) {

boolean d = done;

T v = queue.poll();

boolean empty = v == null;

if (checkTerminated(d, empty)) {

return;

}

if (empty) {

break;

}

subscriber.onNext(v);

e++;

}

if (e == r && checkTerminated(done, queue.isEmpty())) {

break;

}

if (e != 0) {

BackpressureHelper.produced(requested, e);

p.request(e);

}

missed = wip.addAndGet(-missed);

if (missed == 0) {

break;

}

}

}

boolean checkTerminated(boolean d, boolean empty) {

if (subscriber.isUnsubscribed()) {

queue.clear();

return true;

}

if (d) {

Throwable e = error;

if (e != null) {

subscriber.onError(e);

return true;

} else

if (empty) {

subscriber.onCompleted();

return true;

}

}

return false;

}

});

});

By now, the pattern should be quite familiar. We queue up the item or save the exception, then increment the **wip** counter and schedule the draining of the queue. This is necessary as values may arrive the same time the downstream issues a request. Issuing a request has to schedule the drain as well because values may be available in the queue already. The drain loop emits what it can and asks for replenishment from the upstream **Producer** it got through the **setProducer()** call.  
  
Naturally, one can extend this with additional safeguards, error-delay capability, parametric initial request amount and even stable replenishment amounts. This **trySchedule** setup has the property that it doesn't require a single threaded scheduler to begin with as it self-trampolines: due to the **getAndIncrement**, only a single thread will issue the drain task at a time and then only when the **wip** counter is decremented to zero will open the opportunity for scheduling another drain task by somebody.

### Conclusion

In this post, I've tried to clear up the confusion around the **subscribeOn** and **observeOn** operators by showing a simplified, clutter free way of implementing them.  
  
We saw then that the complication in RxJava comes from the need for handling backpressure somewhat transparently for consumers that do or don't directly drive a sequence through it.  
  
Now that the inner workings and structures have been clarified, let's continue with the discussion about operator fusion where I can now use **subscribeOn** and **observeOn** as an example how macro- and micro-fusion can help around the asynchronous boundaries they provide.