# [RxJava Convention](https://wiki.eisgroup.com/display/GRC/RxJava+Convention)

Motivation

Goals of this document are to collect best RxJava practices and to provide a guide that answer two questions:

1. Where to use RxJava?
2. How to use RxJava?

This guide is mostly based on book ["Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications"](https://www.amazon.com/Reactive-Programming-RxJava-Asynchronous-Applications-ebook/dp/B01LZQGIIC/ref=mt_kindle?_encoding=UTF8&me=).

Where to use RxJava

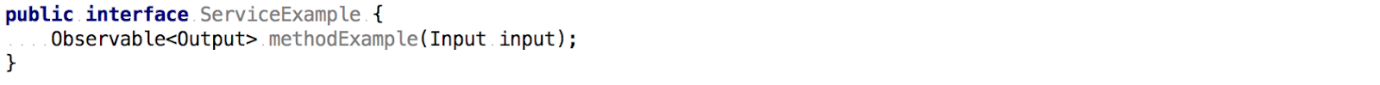
The use cases where one would want to consider using RxJava are described below.

Modules API

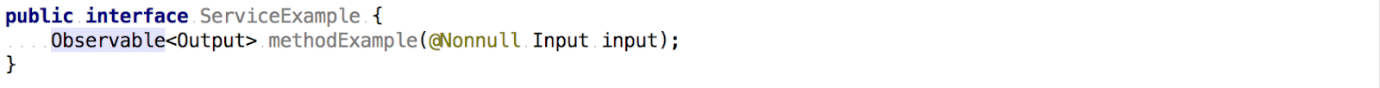
### How to pass input data into the method

When you design parameters of the method, please use these recommendations:

* Define contract for each parameter explicitly by using JSR-305 annotations. It provides two abilities:
  + Verify correctness of client code by automated tools
  + Simplify your implementation. For example, you have the following method in a service:



It means that your contract says: "Any possible value of input is a valid value and I process it without any error". But in most cases, for instance, null is not a valid value and declining to process it is a valid contract. So you can change the method:



and all implementations of the method can assume that clients won't pass null as an input, because otherwise they violate contract of the method and they cannot expect correct behavior.

* If a parameter is a primitive type, don't use wrappers. The problem with wrappers is that they change set of possible values for the type and as result complicates your code. For example, set of possible values for boolean is {true, false} and you can apply binary logic (or Boolean algebra) to a params of such type. But set of possible values for Boolean is {null, True, False} and here you need to apply ternary logic, otherwise sooner or later you will run into bugs.
* Try to avoid passing RxJava monads (like Observable) as a parameter. If fact passing Observable as a param is a legit thing in some situations (for example, when you pass Observable to define virtual discrete time for the method), but for business logic it would be better to avoid such technique due to possible control flow and concurrency bugs.

### How to return result

Use the following recommendations for the results of service's methods:

* Never return **null** .
* Always use some monad-based streaming API. In all examples below we use RxJava monads, but if you use tools with streaming API like Spark, your services can return the tool-specific abstractions.  
  There are recommendations how to select concrete RxJava type:
  + Method does some computation, but it doesn't return any result . Use Completable:



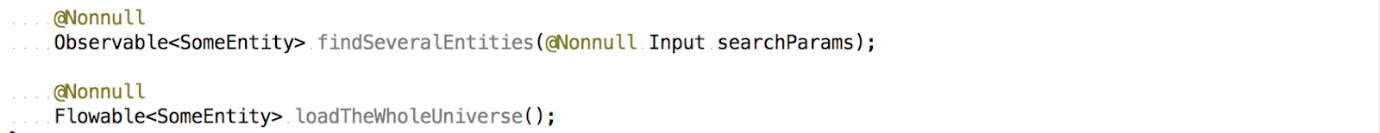
* Method always returns non-nullable single value; there is no case that it completes without any result or returns **null** . Use Single:



* Method can return a value or can complete without any value (like return **null** for blocking methods) . Use Maybe:



* Method can return many values or can complete without any value. If producing of n-th value fails, it won't try to produce (n + 1)-th value, but will fail . Use Observable or Flowable (and use [this guide](https://github.com/ReactiveX/RxJava/wiki/What%27s-different-in-2.0#observable-and-flowable)to choose which of them should be used for your case):



* Method can return many values or can complete without any value. If producing of n-th value fails, it will return error for n-th value and continue producing next values . Use Either with Observable or Flowable. Note: Either is a monad that represents values with two possibilities: a value of type **Either a b**is either **Left a**or **Right b**. It isn't part of JDK like Optional, so it should be either implemented in Platform or used from a 3rd party library :



Integration with external systems

If an external system uses non-reactive API, the system should be isolated by integration layer with reactive API. So use RxJava both for API of the layer and for its implementation.

Callbacks

Reactive programming provides more sophisticated tools for resolving the same design problem that callbacks do, but without side effects like "callback hell". So, if you want to design an API with callbacks, please seriously consider replacing it by reactive API.

Alternative to Streams

RxJava API has more operators than streams making it more flexible. In some cases due to design RxJava has way smaller memory footprint than streams. This is because it processes single event till blocked, in contrary to stream where all are processed at once. Example: Streams.findFirst is slower than Observable.findFirst().

Wrapping heavy operations

Use RxJava for declaration and implementation of methods that execute some heavy staff like file reading, network communication, heavy calculations, etc. Such methods can be composed into a pipeline by flatMap() or merge() that allows to use declarative concurrency when we need performance optimizations. See Use flatMap()/merge() as Asynchronous Chaining Operators for details.

How to use RxJava

This section describes best practices that can be useful in most cases.

Avoid creating an Observable or an ObservableOperator

In business code you usually don't need to create any observable, because all observables should come from API of modules that you are using. But in situation when you need to create it, try to avoid Observable.create() and manual emitting events - use RxJava factory methods instead. The same is about operators - in most cases you can achieve your goals by combining existing operators instead of creating new one.

**Reason:**usually selfmade observables forget about loading and backpressure support, as result they can flood subscribers and causes unexpected crashes in event processing. Implementation of operators also requires more deep knowledge of RxJava contracts and API than most users of RxJava have.

Use fast non-blocking functions in a pipeline

All functions that you pass into a pipeline (by invocation functions like map()) should be non-blocking and fast. In ideal case they should be [pure functions](https://en.wikipedia.org/wiki/Pure_function) .

**Reason:**RxJava creates a single Worker instance for the entire pipeline, mostly to guarantee sequential processing of events. This means that if one of your operators is particularly slow (for example, you map() is reading data from disk in order to transform events passing by) this costly operation will be invoked within the same thread. As a result a single broken operator can slow down the entire pipeline, from production to consumption. This is an antipattern in RxJava, so operators should be nonblocking, fast, and as pure as possible. If you need to do some heavy operation into the pipeline, you need to hide id under RxJava API and chain it into the pipeline by using flatMap(...) or merge(...) (see  Use flatMap()/merge() as Asynchronous Chaining Operators  for details).

Avoid subscribing in domain code

In most cases business logic is just creating Rx pipeline: getting Observables from external API or from factory methods of RxJava, composing them and returning them to some sort of framework or to upper layer. The actual subscription should happen behind the scenes in an infrastructure layer or framework. So it is not a bad practice to call subscribe() yourself, but try to push it out as far as possible.

**Reasons:**it allows to make business code more pure, to use laziness effectively, and to provide optimizations outside of the business code by using declarative concurrency.

Capture All Notifications

It's useful to provide all three arguments to subscribe(). Use Observer interface to group all notification listeners.

**Reason:**it allows to define explicitly how your code reacts on all kind of notifications and to avoid propagating of cryptic exceptions like OnErrorNotImplementedException.

Control your subscriptions

If you subscribe to an Observable, don't forget to unsubscribe from it as soon as you no longer want to receive more events. Use Subscription or Subscriber for that.

**Reason:**this avoids memory leaks and unnecessary load on the system.

Do not pass null to onNext

If you decided to write your own operator or Observable, do not pass null to onNext() .

**Reason:**despite of being a sintacticaly valid event value, **null**will break transformation in most cases, because usually transformation code doesn't expect **null**values. So passing **null**is generally considered non-idiomatic and you should use some kind of replacement instead (like Null Object Pattern or Optional).

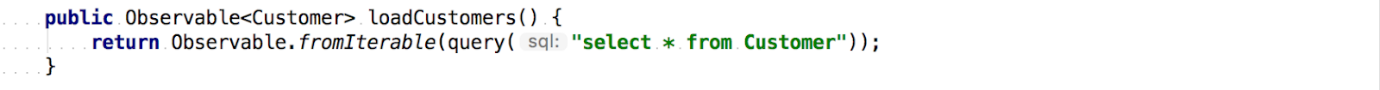
Do not use blocking...() methods in business code

There are third-party APIs that are blocking and there is simply nothing we can do about it. Such APIs should be isolated in integration layer and the layer can use methods like **Observable.blockingGet()** or **Observable.** **blockingFirst()** to integrate that API.

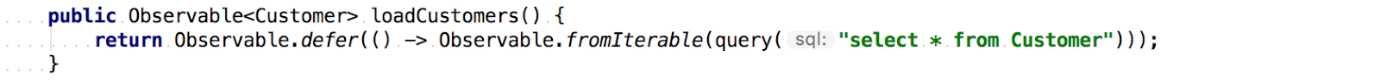
**Reason:**business code should be pure reactive code that is written for async non-blocking execution by default. Having blocking code is a bottleneck that sooner or later will require optimizations that will add unneeded accidental complexity in business code.

Embrace Laziness

When you implement a method that returns Observable, always try to make your implementation lazy. For example, you need to implement a DAO method that loads all customers. First (and bad) implementation can be like this one:



The problem here that we execute heavy select query even if nobody is going to subscribe to the returned observable. To avoid this overhead you can use Observable.defer():



The underlying Observable is still eager, but we have postponed its creation - defer() will wait until someone actually subscribes to it.

**Reason:**laziness allows to defer actual execution of (potentially) heavy methods, like loading/saving data, invoking remote services, etc. In other words - lazy methods don't have any side effects and have almost no performance footprint until actual subscriber has appeared.

Use flatMap()/merge() as Asynchronous Chaining Operators

When you use reactive API, join streams by using flatMap()/merge() instead of subscribing to them.

**Reason:**In business code you need to consider all external Observers as asynchronous and nonblocking. Although in reality all Observables are synchronous by default, such mindset allows to be not opinionated about threading, defaulting to synchronous execution but allowing seamless and almost transparent multithreading when it is needed.

Let's consider an example. Assume we have the following domain entity:

Text

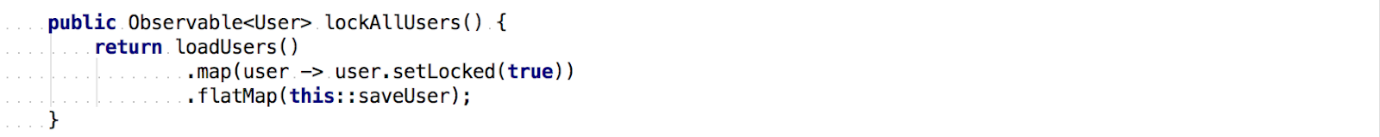
Description automatically generated with medium confidence

And we want to write business code that loads all users, sets locked to true for every user and saves the users. Assume we have the following API to load and save users:

Graphical user interface, text, application

Description automatically generated

So our business method can look like this one:



For example our save/load API uses default Observables and we have three users in our DB. In this case execution time of our method will look like this one:

000173   | main   | Loading users...

001180   | main   | Storing first

002186   | main   | User 'first' is locked

002186   | main   | Storing second

003190   | main   | User 'second' is locked

003191   | main   | Storing third

004192   | main   | User 'third' is locked

You can see that saving is a heavy operation, so we apparently want to use something like Fork-Join framework for saving users: each user can be saved by different thread. And we have decided to add declarative concurrency into saveUser() method to implement that optimization:

A picture containing graphical user interface

Description automatically generated

How will this optimization affect our business code (method lockAllUsers())? In fact it won't affect it at all, because we use flatMap() here. So if saveUser() starts to store each user in different thread, RxJava will automatically fork processing of events concurrently and then join all the results together in the end. After that execution time of our method will look like this:

000031   | main   | Loading users...

001045   | Sched-A-0   | Storing first

001045   | Sched-A-2   | Storing third

001045   | Sched-A-1   | Storing second

002051   | Sched-A-2   | User 'third' is locked

002052   | Sched-A-0   | User 'first' is locked

002053   | Sched-A-0   | User 'second' is locked

So you can see that adding asynchronous processing is transparent for our business code.

Be Mindful of Chained Merge Operators

When operating on default trampoline scheduler one might end up with ***StackOverflowError***on execution simply due to too many chained *merge* operators (including .*mergeWith*, .*concatWith*, .*zipWith*, etc.). While in the simplest of cases this can be avoided by refactoring relevant code to use some combination of .*flatMap* / .fromIterable operators, there will be cases where using merge operators is appropriate. Let us consider the following example where we have an observable-returning-method being called inside .*flatMap*:

where on subscription it fails due to too many .*mergeWith*operators.

We might consider solving this by subscribing on *GenesisScheduler.networking()* , however placement of .*subscribeOn/.observeOn* is important. Placing it outside *.flatMap* operator will have no effect as rxjava operators are single threaded (even if the thread is inherited):

similarly subscribing on networking scheduler will have no effect if done inside the *.flatMap*operator:

the *.subscribeOn*/*.observeOn* will have to invoked at the same level where merging happens in order to fix the issue at hand. In the case above this would be:

All of this is not to say that merge operators should not be used, just that care should be taken when using them especially since RxJava2 prevents the propagation of *StackOverflowError* and so it would not appear in logs.

If you would decide to interact with the test example above, increasing nesting depth will at some point result in *StackOverflowError* in assembly process regardless of used scheduler.

Avoid multithreading in business code

In real life most Observables should come from sources that are asynchronous by their nature, so business code should be written in concurrency-agnostic way: it should assume that it gets async Observables from any API and it shouldn't have any imperative or declarative concurrency code.

**Reason:**concurrency, especially imperative one, significantly increases accidental complexity of business code. Also such code is bug prone and it requires specific skills just to make it works correctly.

Avoid using predefined Schedulers

For multithreading use Genesis predefined schedulers instead of using factory methods from Schedulers class. Nevertheless Schedulers.single() and Schedules.trampoline() still can be used due to theirs single thread nature.

**Reason:**using predefined schedulers can lead to availability issues and unmanageable resource utilization. Also predefined schedulers are used by RxJava for internal purposes, so flooding them can lead to system wide degradation.

There are following issues with each of predefined scheduler:

* Schedulers.newThread() - spawns new thread every time it is requested. Problem: can lead to excessive and uncontrolled resource utilization.
* Schedulers.io() - has unbounded pool of threads. Problem: can lead to excessive and uncontrolled resource utilization.
* Schedulers.computation() - has limited amount of threads, but there are unbounded queues in front of every thread. Problem: can lead to OOM

Using declarative concurrency

**Use subscribeOn() only in infrastructure code**

In mature applications, in terms of Rx adoption, subscribeOn() is very seldom used. Most of the time Observables come from asynchronous sources and they are treated as asynchronous by default. So using subscribeOn() is mostly should be limited by retrofitting existing APIs or libraries in infrastructure code.

An example of using subscribeOn() can be saveUser() method from flapMap() example. In this method we need to save a User in external system, so in fact this method should be placed in integration layer and it's a producer of event that indicates results of user saving. According to Place subscribeOn() and observeOn() correctly we need to place subscribeOn() as near to a Producer as possible, so if we want to add concurrency in user saving, the saveUser() will look like this:

A picture containing graphical user interface

Description automatically generated

**Do not invoke subscribeOn() subsequently**

If you know that some method uses subscribeOn(), don't try to use subscribeOn() one more time.

**Reason:**subscribeOn() that is closest to the original Observable always 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. Also be aware that despite of ignoring all subsequent subscribeOn() invocations, RxJava still uses schedulers from subscribeOn() a short period of time; thus, multiple subscribeOn() are not only ignored, but also introduce small overhead.

Please look at the example below and its execution log to see how second invocation of subscribeOn() is ignored:

Graphical user interface, text, application

Description automatically generated  
000042   | main   | Starting

000145   | main   | Created

000156   | main   | Exiting

000158   | Sched-A-0   | Subscribed

000158   | Sched-A-0   | A-1

000158   | Sched-A-0   | A-2

000159   | Sched-A-0   | A-3

000159   | Sched-A-0   | A-4

000159   | Sched-A-0   | Got A

000159   | Sched-A-0   | B-1

000159   | Sched-A-0   | B-2

000159   | Sched-A-0   | B-3

000160   | Sched-A-0   | B-4

000160   | Sched-A-0   | Got B

000160   | Sched-A-0   | Completed

**Limit using observeOn() in backend code**

**Reason:**observeOn() is especially important for applications with a UI for which we do not want to block the UI event-dispatching thread. On the server, observeOn() is seldom used because the true source of concurrency is built into most Observables.

**Use observeOn() to control downstream concurrency**

subscribeOn() allows choosing which Scheduler will be used to invoke OnSubscribe (lambda expression inside Observable.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(). To see these difference, compare executions of the following examples:

Text

Description automatically generated

000043   | main   | Starting

000142   | main   | Created

000155   | main   | Exiting

000156   | Sched-A-0   | Subscribed

000156   | Sched-A-0   | A-1

000157   | Sched-A-0   | A-2

000157   | Sched-A-0   | Got A

000157   | Sched-A-0   | B-1

000157   | Sched-A-0   | B-2

000157   | Sched-A-0   | Got B

000158   | Sched-A-0   | Completed

Text

Description automatically generated

000042   | main   | Starting

000141   | main   | Created

000187   | main   | Exiting

000193   | Sched-A-0   | Subscribed

000193   | Sched-A-0   | A-1

000194   | Sched-A-0   | B-1

000194   | Sched-B-0   | A-2

000194   | Sched-B-0   | Got A

000194   | Sched-B-0   | B-2

000195   | Sched-B-0   | Got B

000195   | Sched-B-0   | Completed

**Redefine observeOn() when you need it**

If you want to redefine scheduler for downstream events, invoke observeOn().

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. 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().

See the following code and its execution to figure out the difference:

Graphical user interface, text

Description automatically generated

000051   | main   | Starting  
000147   | main   | Created  
000191   | main   | Exiting  
000195   | Sched-A-0   | Subscribed  
000195   | Sched-A-0   | Found 1: A  
000196   | Sched-A-0   | Found 1: B  
000196   | Sched-B-0   | Found 2: A  
000197   | Sched-B-0   | Found 2: B  
000197   | Sched-C-0   | Found 3: A  
000197   | Sched-C-0   | Got 1: A  
000197   | Sched-C-0   | Found 3: B  
000197   | Sched-C-0   | Got 1: B  
000198   | Sched-C-0   | Completed

**Place subscribeOn() and observeOn() correctly**

Place subscribeOn() near Producer and observeOn() near Consumer.

**Reason:**  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. 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().