RxJava: Reactive Extensions for the JVM

RxJava is a Java VM implementation of Reactive Extensions: a library for composing asynchronous and event-based programs by using observable sequences.

It extends the observer pattern to support sequences of data/events and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety and concurrent data structures.

Version 3.x (Javadoc)

- single dependency: Reactive-Streams
- Java 8+ or Android API 21+ required
- Java 8 lambda-friendly API
- Android desugar friendly
- fixed API mistakes and many limits of RxJava 2
- intended to be a replacement for RxJava 2 with relatively few binary incompatible changes
- non-opinionated about the source of concurrency (threads, pools, event loops, fibers, actors, etc.)
- async or synchronous execution
- virtual time and schedulers for parameterized concurrency
- test and diagnostic support via test schedulers, test consumers and plugin hooks

Learn more about RxJava in general on the Wiki Home.

:information_source: Please read the What's different in 3.0 for details on the changes and migration information when upgrading from 2.x.

Version 2.x

The 2.x version is end-of-life as of **February 28, 2021**. No further development, support, maintenance, PRs and updates will happen. The Javadoc of the very last version, **2.2.21**, will remain accessible.

Version 1.x

The 1.x version is end-of-life as of **March 31, 2018**. No further development, support, maintenance, PRs and updates will happen. The Javadoc of the very last version, **1.3.8**, will remain accessible.

Getting started

Setting up the dependency

The first step is to include RxJava 3 into your project, for example, as a Gradle compile dependency:

```
implementation "io.reactivex.rxjava3:rxjava3:3.x.y"
```

(Please replace `x` and `y` with the latest version numbers: 

)

Hello World

The second is to write the **Hello World** program:

```
package rxjava.examples;

import io.reactivex.rxjava3.core.*;

public class HelloWorld {
    public static void main(String[] args) {
        Flowable.just("Hello world").subscribe(System.out::println);
    }
}
```

Note that RxJava 3 components now live under `io.reactivex.rxjava3` and the base classes and interfaces live under `io.reactivex.rxjava3.core`.

Base classes

RxJava 3 features several base classes you can discover operators on:

- `io.reactivex.rxjava3.core.Flowable` : 0..N flows, supporting Reactive-Streams and backpressure
- `io.reactivex.rxjava3.core.Observable` : 0..N flows, no backpressure,
- `io.reactivex.rxjava3.core.Single` : a flow of exactly 1 item or an error,
- `io.reactivex.rxjava3.core.Completable` : a flow without items but only a completion or error signal,
- `io.reactivex.rxjava3.core.Maybe` : a flow with no items, exactly one item or an error.

Some terminology

Upstream, downstream

The dataflows in RxJava consist of a source, zero or more intermediate steps followed by a data consumer or combinator step (where the step is responsible to consume the dataflow by some means):

```
source.operator1().operator2().operator3().subscribe(consumer);

source.flatMap(value -> source.operator1().operator2().operator3());
```

Here, if we imagine ourselves on `operator2`, looking to the left towards the source is called the **upstream**. Looking to the right towards the subscriber/consumer is called the **downstream**. This is often more apparent when each element is written on a separate line:

```
source
  .operator1()
  .operator2()
  .operator3()
  .subscribe(consumer)
```

Objects in motion

In RxJava's documentation, **emission**, **emits**, **item**, **event**, **signal**, **data** and **message** are considered synonyms and represent the object traveling along the dataflow.

Backpressure

When the dataflow runs through asynchronous steps, each step may perform different things with different speed. To avoid overwhelming such steps, which usually would manifest itself as increased memory usage due to temporary buffering or the need for skipping/dropping data, so-called backpressure is applied, which is a form of flow control where the steps can express how many items are they ready to process. This allows constraining the memory usage of the dataflows in situations where there is generally no way for a step to know how many items the upstream will send to it.

In RxJava, the dedicated `Flowable` class is designated to support backpressure and `Observable` is dedicated to the non-backpressured operations (short sequences, GUI interactions, etc.). The other types, `Single`, `Maybe` and `Completable` don't support backpressure nor should they; there is always room to store one item temporarily.

Assembly time

The preparation of dataflows by applying various intermediate operators happens in the so-called **assembly time**:

```
Flowable<Integer> flow = Flowable.range(1, 5)
.map(v -> v * v)
.filter(v -> v % 3 == 0)
;
```

At this point, the data is not flowing yet and no side-effects are happening.

Subscription time

This is a temporary state when `subscribe()` is called on a flow that establishes the chain of processing steps internally:

```
flow.subscribe(System.out::println)
```

This is when the **subscription side-effects** are triggered (see `doOnSubscribe`). Some sources block or start emitting items right away in this state.

Runtime

This is the state when the flows are actively emitting items, errors or completion signals:

```
Observable.create(emitter -> {
     while (!emitter.isDisposed()) {
         long time = System.currentTimeMillis();
         emitter.onNext(time);
         if (time % 2 != 0) {
             emitter.onError(new IllegalStateException("Odd millisecond!"));
             break;
         }
     }
})
.subscribe(System.out::println, Throwable::printStackTrace);
```

Practically, this is when the body of the given example above executes.

Simple background computation

One of the common use cases for RxJava is to run some computation, network request on a background thread and show the results (or error) on the UI thread:

```
import io.reactivex.rxjava3.schedulers.Schedulers;

Flowable.fromCallable(() -> {
    Thread.sleep(1000); //  imitate expensive computation
    return "Done";
})
  .subscribeOn(Schedulers.io())
  .observeOn(Schedulers.single())
  .subscribe(System.out::println, Throwable::printStackTrace);

Thread.sleep(2000); // <--- wait for the flow to finish
```

This style of chaining methods is called a **fluent API** which resembles the **builder pattern**. However, RxJava's reactive types are immutable; each of the method calls returns a new `Flowable` with added behavior. To illustrate, the example can be rewritten as follows:

```
Flowable<String> source = Flowable.fromCallable(() -> {
    Thread.sleep(1000); //  imitate expensive computation
    return "Done";
});

Flowable<String> runBackground = source.subscribeOn(Schedulers.io());

Flowable<String> showForeground = runBackground.observeOn(Schedulers.single());

showForeground.subscribe(System.out::println, Throwable::printStackTrace);

Thread.sleep(2000);
```

Typically, you can move computations or blocking IO to some other thread via `subscribeOn`. Once the data is ready, you can make sure they get processed on the foreground or GUI thread via `observeOn`.

Schedulers

RxJava operators don't work with `Thread`s or `ExecutorService`s directly but with so-called `Scheduler`s that abstract away sources of concurrency behind a uniform API. RxJava 3 features several standard schedulers accessible via `Schedulers` utility class.

- `Schedulers.computation()`: Run computation intensive work on a fixed number of dedicated threads in the background. Most asynchronous operators use this as their default `Scheduler`.
- `Schedulers.io()`: Run I/O-like or blocking operations on a dynamically changing set of threads.
- `Schedulers.single()`: Run work on a single thread in a sequential and FIFO manner.
- `Schedulers.trampoline()`: Run work in a sequential and FIFO manner in one of the participating threads, usually for testing purposes.

These are available on all JVM platforms but some specific platforms, such as Android, have their own typical `Scheduler`s defined: `AndroidSchedulers.mainThread()`, `SwingScheduler.instance()` or `JavaFXSchedulers.gui()`.

In addition, there is an option to wrap an existing `Executor` (and its subtypes such as `ExecutorService`) into a `Scheduler` via `Schedulers.from(Executor)`. This can be used, for example, to have a larger but still fixed pool of threads (unlike `computation()` and `io()` respectively).

The `Thread.sleep(2000);` at the end is no accident. In RxJava the default `Scheduler`s run on daemon threads, which means once the Java main thread exits, they all get stopped and background computations may never happen. Sleeping for some time in this example situations lets you see the output of the flow on the console with time to spare.

Concurrency within a flow

Flows in RxJava are sequential in nature split into processing stages that may run **concurrently** with each other:

```
Flowable.range(1, 10)
  .observeOn(Schedulers.computation())
  .map(v -> v * v)
  .blockingSubscribe(System.out::println);
```

This example flow squares the numbers from 1 to 10 on the **computation** `Scheduler` and consumes the results on the "main" thread (more precisely, the caller thread of `blockingSubscribe`). However, the lambda `v -> v * v` doesn't run in parallel for this flow; it receives the values 1 to 10 on the same computation thread one after the other.

Parallel processing

Processing the numbers 1 to 10 in parallel is a bit more involved:

```
Flowable.range(1, 10)
  .flatMap(v ->
      Flowable.just(v)
        .subscribeOn(Schedulers.computation())
        .map(w -> w * w)
  )
  .blockingSubscribe(System.out::println);
```

Practically, parallelism in RxJava means running independent flows and merging their results back into a single flow. The operator `flatMap` does this by first mapping each number from 1 to 10 into its own individual `Flowable`, runs them and merges the computed squares.

Note, however, that `flatMap` doesn't guarantee any order and the items from the inner flows may end up interleaved. There are alternative operators:

- `concatMap` that maps and runs one inner flow at a time and
- `concatMapEager` which runs all inner flows "at once" but the output flow will be in the order those inner flows were created.

Alternatively, the `Flowable.parallel()` operator and the `ParallelFlowable` type help achieve the same parallel processing pattern:

```
Flowable.range(1, 10)
  .parallel()
  .runOn(Schedulers.computation())
  .map(v -> v * v)
  .sequential()
  .blockingSubscribe(System.out::println);
```

Dependent sub-flows

`flatMap` is a powerful operator and helps in a lot of situations. For example, given a service that returns a `Flowable`, we'd like to call another service with values emitted by the first service:

```
Flowable<Inventory> inventorySource = warehouse.getInventoryAsync();

inventorySource
    .flatMap(inventoryItem -> erp.getDemandAsync(inventoryItem.getId())
            .map(demand -> "Item " + inventoryItem.getName() + " has demand " +
demand))
    .subscribe(System.out::println);
```

Continuations

Sometimes, when an item has become available, one would like to perform some dependent computations on it. This is sometimes called **continuations** and, depending on what should happen and what types are involved, may involve various operators to accomplish.

Dependent

The most typical scenario is to given a value, invoke another service, await and continue with its result:

```
service.apiCall()
.flatMap(value -> service.anotherApiCall(value))
.flatMap(next -> service.finalCall(next))
```

It is often the case also that later sequences would require values from earlier mappings. This can be achieved by moving the outer `flatMap` into the inner parts of the previous `flatMap` for example:

```
service.apiCall()
.flatMap(value ->
    service.anotherApiCall(value)
    .flatMap(next -> service.finalCallBoth(value, next))
)
```

Here, the original `value` will be available inside the inner `flatMap`, courtesy of lambda variable capture.

Non-dependent

In other scenarios, the result(s) of the first source/dataflow is irrelevant and one would like to continue with a quasi independent another source. Here, `flatMap` works as well:

```
Observable continued = sourceObservable.flatMapSingle(ignored -> someSingleSource)
continued.map(v -> v.toString())
  .subscribe(System.out::println, Throwable::printStackTrace);
```

however, the continuation in this case stays `Observable` instead of the likely more appropriate `Single`. (This is understandable because from the perspective of `flatMapSingle`, `sourceObservable` is a multi-valued source and thus the mapping may result in multiple values as well).

Often though there is a way that is somewhat more expressive (and also lower overhead) by using `Completable` as the mediator and its operator `andThen` to resume with something else:

```
sourceObservable
  .ignoreElements()           // returns Completable
  .andThen(someSingleSource)
  .map(v -> v.toString())
```

The only dependency between the `sourceObservable` and the `someSingleSource` is that the former should complete normally in order for the latter to be consumed.

Deferred-dependent

Sometimes, there is an implicit data dependency between the previous sequence and the new sequence that, for some reason, was not flowing through the "regular channels". One would be inclined to write such continuations as follows:

```
AtomicInteger count = new AtomicInteger();

Observable.range(1, 10)
  .doOnNext(ignored -> count.incrementAndGet())
  .ignoreElements()
  .andThen(Single.just(count.get()))
  .subscribe(System.out::println);
```

Unfortunately, this prints `0` because `Single.just(count.get())` is evaluated at **assembly time** when the dataflow hasn't even run yet. We need something that defers the evaluation of this `Single` source until **runtime** when the main source completes:

```
AtomicInteger count = new AtomicInteger();

Observable.range(1, 10)
  .doOnNext(ignored -> count.incrementAndGet())
  .ignoreElements()
  .andThen(Single.defer(() -> Single.just(count.get())))
  .subscribe(System.out::println);
```

or

```
AtomicInteger count = new AtomicInteger();
```

```
Observable.range(1, 10)
  .doOnNext(ignored -> count.incrementAndGet())
  .ignoreElements()
  .andThen(Single.fromCallable(() -> count.get()))
  .subscribe(System.out::println);
```

Type conversions

Sometimes, a source or service returns a different type than the flow that is supposed to work with it. For example, in the inventory example above, `getDemandAsync` could return a `Single<DemandRecord>`. If the code example is left unchanged, this will result in a compile-time error (however, often with a misleading error message about lack of overload).

In such situations, there are usually two options to fix the transformation: 1) convert to the desired type or 2) find and use an overload of the specific operator supporting the different type.

Converting to the desired type

Each reactive base class features operators that can perform such conversions, including the protocol conversions, to match some other type. The following matrix shows the available conversion options:

	Flowable	Observable	Single	Maybe	Completable
Flowable		`toObservable`	`first`, `firstOrError`, `single`, `singleOrError`, `last`, `lastOrError`[1]	`firstElement`, `singleElement`, `lastElement`	`ignoreElements`
Observable	`toFlowable`[2]		`first`, `firstOrError`, `single`, `singleOrError`, `last`, `lastOrError`[1]	`firstElement`, `singleElement`, `lastElement`	`ignoreElements`
Single	`toFlowable`[3]	`toObservable`		`toMaybe`	`ignoreElement`
Maybe	`toFlowable`[3]	`toObservable`	`toSingle`		`ignoreElement`
Completable	`toFlowable`	`toObservable`	`toSingle`	`toMaybe`	

[1]: When turning a multi-valued source into a single-valued source, one should decide which of the many source values should be considered as the result.

[2]: Turning an `Observable` into `Flowable` requires an additional decision: what to do with the potential unconstrained flow of the source `Observable`? There are several strategies available (such as buffering, dropping, keeping the latest) via the `BackpressureStrategy` parameter or via standard `Flowable` operators such as `onBackpressureBuffer`, `onBackpressureDrop`, `onBackpressureLatest` which also allow further customization of the backpressure behavior.

[3]: When there is only (at most) one source item, there is no problem with backpressure as it can be always stored until the downstream is ready to consume.

Using an overload with the desired type

Many frequently used operator has overloads that can deal with the other types. These are usually named with the suffix of the target type:

Operator	Overloads
`flatMap`	`flatMapSingle`, `flatMapMaybe`, `flatMapCompletable`, `flatMapIterable`
`concatMap`	`concatMapSingle`, `concatMapMaybe`, `concatMapCompletable`, `concatMapIterable`
`switchMap`	`switchMapSingle`, `switchMapMaybe`, `switchMapCompletable`

The reason these operators have a suffix instead of simply having the same name with different signature is type erasure. Java doesn't consider signatures such as `operator(Function<T, Single<R>>)` and `operator(Function<T, Maybe<R>>)` different (unlike C#) and due to erasure, the two `operator`s would end up as duplicate methods with the same signature.

Operator naming conventions

Naming in programming is one of the hardest things as names are expected to be not long, expressive, capturing and easily memorable. Unfortunately, the target language (and pre-existing conventions) may not give too much help in this regard (unusable keywords, type erasure, type ambiguities, etc.).

Unusable keywords

In the original Rx.NET, the operator that emits a single item and then completes is called `Return(T)`. Since the Java convention is to have a lowercase letter start a method name, this would have been `return(T)` which is a keyword in Java and thus not available. Therefore, RxJava chose to name this operator `just(T)`. The same limitation exists for the operator `Switch`, which had to be named `switchOnNext`. Yet another example is `Catch` which was named `onErrorResumeNext`.

Type erasure

Many operators that expect the user to provide some function returning a reactive type can't be overloaded because the type erasure around a `Function<T, X>` turns such method signatures into duplicates. RxJava chose to name such operators by appending the type as suffix as well:

```
Flowable<R> flatMap(Function<? super T, ? extends Publisher<? extends R>> mapper)

Flowable<R> flatMapMaybe(Function<? super T, ? extends MaybeSource<? extends R>>
mapper)
```

Type ambiguities

Even though certain operators have no problems from type erasure, their signature may turn up being ambiguous, especially if one uses Java 8 and lambdas. For example, there are several overloads of `concatWith` taking the various other reactive base types as arguments (for providing convenience and performance benefits in the underlying implementation):

```
Flowable<T> concatWith(Publisher<? extends T> other);

Flowable<T> concatWith(SingleSource<? extends T> other);
```

Both `Publisher` and `SingleSource` appear as functional interfaces (types with one abstract method) and may encourage users to try to provide a lambda expression:

```
someSource.concatWith(s -> Single.just(2))
.subscribe(System.out::println, Throwable::printStackTrace);
```

Unfortunately, this approach doesn't work and the example does not print `2` at all. In fact, since version 2.1.10, it doesn't even compile because at least 4 `concatWith` overloads exist and the compiler finds the code above ambiguous.

The user in such situations probably wanted to defer some computation until the `someSource` has completed, thus the correct unambiguous operator should have been `defer`:

```
someSource.concatWith(Single.defer(() -> Single.just(2)))
.subscribe(System.out::println, Throwable::printStackTrace);
```

Sometimes, a suffix is added to avoid logical ambiguities that may compile but produce the wrong type in a flow:

```
Flowable<T> merge(Publisher<? extends Publisher<? extends T>> sources);

Flowable<T> mergeArray(Publisher<? extends T>... sources);
```

This can get also ambiguous when functional interface types get involved as the type argument `T`.

Error handling

Dataflows can fail, at which point the error is emitted to the consumer(s). Sometimes though, multiple sources may fail at which point there is a choice whether or not wait for all of them to complete or fail. To indicate this opportunity, many operator names are suffixed with the `DelayError` words (while others feature a `delayError` or `delayErrors` boolean flag in one of their overloads):

```
Flowable<T> concat(Publisher<? extends Publisher<? extends T>> sources);

Flowable<T> concatDelayError(Publisher<? extends Publisher<? extends T>> sources);
```

Of course, suffixes of various kinds may appear together:

```
Flowable<T> concatArrayEagerDelayError(Publisher<? extends T>... sources);
```

Base class vs base type

The base classes can be considered heavy due to the sheer number of static and instance methods on them. RxJava 3's design was heavily influenced by the Reactive Streams specification, therefore, the library features a class and an interface per each reactive type:

Type	Class	Interface	Consumer
0..N backpressured	`Flowable`	`Publisher`[1]	`Subscriber`
0..N unbounded	`Observable`	`ObservableSource`[2]	`Observer`
1 element or error	`Single`	`SingleSource`	`SingleObserver`
0..1 element or error	`Maybe`	`MaybeSource`	`MaybeObserver`
0 element or error	`Completable`	`CompletableSource`	`CompletableObserver`

[1]The `org.reactivestreams.Publisher` is part of the external Reactive Streams library. It is the main type to interact with other reactive libraries through a standardized mechanism governed by the Reactive Streams specification.

[2]The naming convention of the interface was to append `Source` to the semi-traditional class name. There is no `FlowableSource` since `Publisher` is provided by the Reactive Streams library (and subtyping it wouldn't have helped with interoperation either). These interfaces are, however, not standard in the sense of the Reactive Streams specification and are currently RxJava specific only.

R8 and ProGuard settings

By default, RxJava itself doesn't require any ProGuard/R8 settings and should work without problems. Unfortunately, the Reactive Streams dependency since version 1.0.3 has embedded Java 9 class files in its JAR that can cause warnings with the plain ProGuard:

```
Warning: org.reactivestreams.FlowAdapters$FlowPublisherFromReactive: can't find
superclass or interface java.util.concurrent.Flow$Publisher
Warning: org.reactivestreams.FlowAdapters$FlowToReactiveProcessor: can't find
superclass or interface java.util.concurrent.Flow$Processor
Warning: org.reactivestreams.FlowAdapters$FlowToReactiveSubscriber: can't find
superclass or interface java.util.concurrent.Flow$Subscriber
Warning: org.reactivestreams.FlowAdapters$FlowToReactiveSubscription: can't find
superclass or interface java.util.concurrent.Flow$Subscription
Warning: org.reactivestreams.FlowAdapters: can't find referenced class
java.util.concurrent.Flow$Publisher
```

It is recommended one sets up the following `-dontwarn` entry in the application's `proguard-ruleset` file:

```
-dontwarn java.util.concurrent.Flow*
```

For R8, the RxJava jar includes the `META-INF/proguard/rxjava3.pro` with the same no-warning clause and should apply automatically.

Further reading

For further details, consult the wiki.

Communication

- Google Group: RxJava
- Twitter: @RxJava

- GitHub Issues
- StackOverflow: rx-java and rx-java2
- Gitter.im

Versioning

Version 3.x is in development. Bugfixes will be applied to both 2.x and 3.x branches, but new features will only be added to 3.x.

Minor 3.x increments (such as 3.1, 3.2, etc) will occur when non-trivial new functionality is added or significant enhancements or bug fixes occur that may have behavioral changes that may affect some edge cases (such as dependence on behavior resulting from a bug). An example of an enhancement that would classify as this is adding reactive pull backpressure support to an operator that previously did not support it. This should be backwards compatible but does behave differently.

Patch 3.x.y increments (such as 3.0.0 -> 3.0.1, 3.3.1 -> 3.3.2, etc) will occur for bug fixes and trivial functionality (like adding a method overload). New functionality marked with an `@Beta` or `@Experimental` annotation can also be added in the patch releases to allow rapid exploration and iteration of unstable new functionality.

@Beta

APIs marked with the `@Beta` annotation at the class or method level are subject to change. They can be modified in any way, or even removed, at any time. If your code is a library itself (i.e. it is used on the CLASSPATH of users outside your control), you should not use beta APIs, unless you repackage them (e.g. using ProGuard, shading, etc).

@Experimental

APIs marked with the `@Experimental` annotation at the class or method level will almost certainly change. They can be modified in any way, or even removed, at any time. You should not use or rely on them in any production code. They are purely to allow broad testing and feedback.

@Deprecated

APIs marked with the `@Deprecated` annotation at the class or method level will remain supported until the next major release, but it is recommended to stop using them.

io.reactivex.rxjava3.internal.*

All code inside the `io.reactivex.rxjava3.internal.*` packages are considered private API and should not be relied upon at all. It can change at any time.

Full Documentation

- Wiki
- Javadoc
- Latest snaphot Javadoc
- Javadoc of a specific release version: `http://reactivex.io/RxJava/3.x/javadoc/3.x.y/`

Binaries

Binaries and dependency information for Maven, Ivy, Gradle and others can be found at http://search.maven.org.

Example for Gradle:

```
implementation 'io.reactivex.rxjava3:rxjava:x.y.z'
```

and for Maven:

```
<dependency>
    <groupId>io.reactivex.rxjava3</groupId>
    <artifactId>rxjava</artifactId>
    <version>x.y.z</version>
</dependency>
```

and for Ivy:

```
<dependency org="io.reactivex.rxjava3" name="rxjava" rev="x.y.z" />
```

Snapshots

Snapshots after May 1st, 2021 are available via
https://oss.sonatype.org/content/repositories/snapshots/io/reactivex/rxjava3/rxjava/

```
repositories {
  maven { url 'https://oss.sonatype.org/content/repositories/snapshots' }
}

dependencies {
  implementation 'io.reactivex.rxjava3:rxjava:3.0.0-SNAPSHOT'
}
```

Snapshots before May 1st, 2021 are available via https://oss.jfrog.org/libs-snapshot/io/reactivex/rxjava3/rxjava/
(Note that due to the Sunset of Bintray, our jfrog access has been severed, hence the new snapshot repo above.)

```
repositories {
    maven { url 'https://oss.jfrog.org/libs-snapshot' }
}

dependencies {
    implementation 'io.reactivex.rxjava3:rxjava:3.0.0-SNAPSHOT'
}
```

JavaDoc snapshots are available at http://reactivex.io/RxJava/3.x/javadoc/snapshot

Build

To build:

```
$ git clone git@github.com:ReactiveX/RxJava.git
$ cd RxJava/
$ ./gradlew build
```

Further details on building can be found on the Getting Started page of the wiki.

Bugs and Feedback

For bugs, questions and discussions please use the Github Issues.

LICENSE