Table of contents

Introduction

Welcome to the new major release of RxJava, a library for composing asynchronous and event-based programs using observable sequences for the Java VM.

As with every such release, there have been quite a lot of trivial and non-trivial changes, cleanups and improvements all across the codebase, which warrant some detailed and comprehensive explanations nonetheless.

With each major release, we take the liberty to introduce potential and actual binary and behavioral incompatible changes so that past mistakes can be corrected and technical debt can be repaid.

Please read this guide to its full extent before posting any issue about "why X no longer compiles". Please also take note of sentences marked with :warning: indicating some migration pitfalls. Information about related discussions and the code changes themselves can be found in each section under the :information_source: **Further references:** marker.

2.x now in maintenance mode

With the release of RxJava 3.0.0, the previous version line, 2.2.x, is in maintenance mode. This means **only bugfixes** will be accepted and applied; **no new operators** or **documentation changes** will be accepted or applied.

:information_source: 2.x will be supported until **February 28, 2021**, after which all development on that branch will stop.

Maven coordinates

RxJava 3 lives in the group `io.reactivex.rxjava3` with artifact ID `rxjava`. Official language/platform adaptors will also be located under the group `io.reactivex.rxjava3`.

The following examples demonstrate the typical import statements. Please consider the latest version and replace `3.0.0` with the numbers from the badge: maven central 3.1.5

Gradle import

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

Maven import

```
<dependency>
  <groupId>io.reactivex.rxjava3</groupId>
  <artifactId>rxjava</artifactId>
  <version>3.0.0</version>
</dependency>
```

:information_source: **Further references:** PR #6421

JavaDocs

The 3.x documentation of the various components can be found at

- http://reactivex.io/RxJava/3.x/javadoc/

Sub-version specific documentation is available under a version tag, for example

- http://reactivex.io/RxJava/3.x/javadoc/3.0.0-RC9

(replace `3.0.0-RC9` with the numbers from the badge: maven central 3.1.5).

The documentation of the current snapshot is under

- http://reactivex.io/RxJava/3.x/javadoc/snapshot

Java 8

For a long time, RxJava was limited to Java 6 API due to how Android was lagging behind in its runtime support. This changed with the upcoming Android Studio 4 previews where a process called desugaring is able to turn many Java 7 and 8 features into Java 6 compatible ones transparently.

This allowed us to increase the baseline of RxJava to Java 8 and add official support for many Java 8 constructs:

- `Stream`: use `java.util.stream.Stream` as a source or expose sequences as **blocking** `Stream`s.
- Stream `Collector`s: aggregate items into collections specified by standard transformations.
- `Optional`: helps with the **non-null**ness requirement of RxJava
- `CompletableFuture`: consume `CompletableFuture`s non-blockingly or expose single results as `CompletableFuture`s.
- Use site non-null annotation: helps with some functional types be able to return null in specific circumstances.

However, some features won't be supported:

- `java.time.Duration`: would add a lot of overloads; can always be decomposed into the `time` + `unit` manually.
- `java.util.function`: these can't throw `Throwable`s, overloads would create bloat and/or ambiguity

Consequently, one has to change the project's compilation target settings to Java 8:

```
sourceCompatibility = JavaVersion.VERSION_1_8
targetCompatibility = JavaVersion.VERSION_1_8
```

or

```
android {
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
```

⚠ **Note on the internal Java 8 support**

Due to the state of the Android Desugar tooling, as of writing this page, the internals of pre-existing, non-Java 8 related RxJava operators do not use Java 8 constructs or types. This allows using these "older" operators with Android API levels where the desugaring tool doesn't provide automatic Java 8 backports of various constructs.

:information_source: **Further references:** Issue #6695, PR #6765, other PRs

Package structure

RxJava 3 components are located under the `io.reactivex.rxjava3` package (RxJava 1 has `rx` and RxJava 2 is just `io.reactivex`. This allows version 3 to live side by side with the earlier versions. In addition, the core types of RxJava (`Flowable` , `Observer` , etc.) have been moved to `io.reactivex.rxjava3.core` .

Component	RxJava 2	RxJava 3
Core	`io.reactivex`	`io.reactivex.rxjava3.core`
Annotations	`io.reactivex.annotations`	`io.reactivex.rxjava3.annotations`
Disposables	`io.reactivex.disposables`	`io.reactivex.rxjava3.disposables`
Exceptions	`io.reactivex.exceptions`	`io.reactivex.rxjava3.exceptions`
Functions	`io.reactivex.functions`	`io.reactivex.rxjava3.functions`
Flowables	`io.reactivex.flowables`	`io.reactivex.rxjava3.flowables`
Observables	`io.reactivex.observables`	`io.reactivex.rxjava3.observables`
Subjects	`io.reactivex.subjects`	`io.reactivex.rxjava3.subjects`
Processors	`io.reactivex.processors`	`io.reactivex.rxjava3.processors`
Observers	`io.reactivex.observers`	`io.reactivex.rxjava3.observers`
Subscribers	`io.reactivex.subscribers`	`io.reactivex.rxjava3.subscribers`
Parallel	`io.reactivex.parallel`	`io.reactivex.rxjava3.parallel`
Internal	`io.reactivex.internal`	`io.reactivex.rxjava3.internal`

⚠ **Note on running "Organize Imports" from the IDE**

Due to naming matches, IDE's tend to import `java.util.Observable` instead of picking RxJava's `io.reactivex.rxjava3.core.Observable` . One can usually have the IDE ignore `java.util.Observable` and `java.util.Observer` , or otherwise, specify an explicit `import io.reactivex.rxjava3.core.Observable;` in the affected files.

Also since RxJava 3 now requires a Java 8 runtime, the standard library functional interfaces, such as `java.util.function.Function` , may be picked instead of `io.reactivex.rxjava3.functions.Function` . IDEs tend to give non-descriptive errors such as "Function can't be converted to Function", omitting the fact about the package differences.

:information_source: **Further references:** PR #6621

Behavior changes

Sometimes, the design of components and operators turn out to be inadequate, too limited or wrong in some circumstances. Major releases such as this allow us to make the necessary changes that would have caused all sorts of problems in a patch release.

More undeliverable errors

With RxJava 2.x, the goal was set to not let any errors slip away in case the sequence is no longer able to deliver them to the consumers for some reason. Despite our best efforts, errors still could have been lost in various race conditions across many dozen operators.

Fixing this in a 2.x patch would have caused too much trouble, therefore, the fix was postponed to the, otherwise already considerably changing, 3.x release. Now, canceling an operator that delays errors internally will signal those errors to the global error handler via `RxJavaPlugins.onError()` .

undeliverable-example

```
RxJavaPlugins.setErrorHandler(error -> System.out.println(error));

PublishProcessor<Integer> main = PublishProcessor.create();
PublishProcessor<Integer> inner = PublishProcessor.create();

// switchMapDelayError will delay all errors
TestSubscriber<Integer> ts = main.switchMapDelayError(v -> inner).test();

main.onNext(1);

// the inner fails
inner.onError(new IOException());

// the consumer is still clueless
ts.assertEmpty();

// the consumer cancels
ts.cancel();

// console prints
// io.reactivex.rxjava3.exceptions.UndeliverableException:
// The exception could not be delivered to the consumer because
// it has already canceled/disposed the flow or the exception has
// nowhere to  go to begin with. Further reading:
// https://github.com/ReactiveX/RxJava/wiki/What's-different-in-2.0#error-handling
// | java.io.IOException
```

:information_source: **Further references:** PRs

Connectable source reset

The purpose of the connectable types (`ConnectableFlowable` and `ConnectableObservable`) is to allow one or more consumers to be prepared before the actual upstream gets streamed to them upon calling `connect()` . This worked correctly for the first time, but had some trouble if the upstream terminated instead of getting disconnected. In this terminating case, depending on whether the connectable was created with `replay()` or `publish()` , fresh consumers would either be unable to receive items from a new connection or would miss items altogether.

With 3.x, connectables have to be reset explicitly when they terminate. This extra step allows consumers to receive cached items or be prepared for a fresh connection.

publish-reset example

With `publish` , if the connectable terminates, consumers subscribing later will only receive the terminal event. One has to call `reset()` so that a late consumer will receive items from a fresh connection.

```
ConnectableFlowable<Integer> connectable = Flowable.range(1, 10).publish();

// prepare consumers, nothing is signaled yet
connectable.subscribe(/* ... */);
connectable.subscribe(/* ... */);

// connect, current consumers will receive items
connectable.connect();

// let it terminate
Thread.sleep(2000);

// late consumers now will receive a terminal event
```

```
connectable.subscribe(
    item -> { },
    error -> { },
    () -> System.out.println("Done!"));

// reset the connectable to appear fresh again
connectable.reset();

// fresh consumers, they will also be ready to receive
connectable.subscribe(
    System.out::println,
    error -> { },
    () -> System.out.println("Done!")
);

// connect, the fresh consumer now gets the new items
connectable.connect();
```

replay-reset example

With `replay`, if the connectable terminates, consumers subscribing later will receive the cached items. One has to call `reset` to discard this cache so that late consumers can then receive fresh items.

```
ConnectableFlowable<Integer> connectable = Flowable.range(1, 10).replay();

// prepare consumers, nothing is signaled yet
connectable.subscribe(System.out::println);
connectable.subscribe(System.out::println);

// connect, current consumers will receive items
connectable.connect();

// let it terminate
Thread.sleep(2000);

// late consumers will still receive the cached items
connectable.subscribe(
    System.out::println,
    error -> { },
    () -> System.out.println("Done!"));

// reset the connectable to appear fresh again
connectable.reset();

// fresh consumers, they will also be ready to receive
connectable.subscribe(
    System.out::println,
    error -> { },
    () -> System.out.println("Done!")
);

// connect, the fresh consumer now gets the new items
connectable.connect();
```

:information_source: **Further references:** Issue #5628, PR #6519

Flowable.publish pause

The implementation of `Flowable.publish` hosts an internal queue to support backpressure from its downstream. In 2.x, this queue, and consequently the upstream source, was slowly draining on its own if all of the resulting `ConnectableFlowable`'s consumers have cancelled. This caused unexpected item loss when the lack of consumers was only temporary.

With 3.x, the implementation pauses and items already in the internal queue will be immediately available to consumers subscribing a bit later.

publish-pause example

```
ConnectableFlowable<Integer> connectable = Flowable.range(1, 200).publish();

connectable.connect();

// the first consumer takes only 50 items and cancels
connectable.take(50).test().assertValueCount(50);

// with 3.x, the remaining items will be still available
connectable.test().assertValueCount(150);
```

:information_source: **Further references:** Issue #5899, PR #6519

Processor.offer null-check

Calling `PublishProcessor.offer()`, `BehaviorProcessor.offer()` or `MulticastProcessor.offer` with a null argument now throws a `NullPointerException` instead of signaling it via `onError` and thus terminating the processor. This now matches the behavior of the `onNext` method required by the Reactive Streams specification.

offer-example

```
PublishProcessor<Integer> pp = PublishProcessor.create();

TestSubscriber<Integer> ts = pp.test();

try {
   pp.offer(null);
} catch (NullPointerException expected) {
}

// no error received
ts.assertEmpty();

pp.offer(1);

// consumers are still there to receive proper items
ts.asssertValuesOnly(1);
```

:information_source: **Further references:** PR #6799

MulticastProcessor.offer fusion-check

`MulticastProcessor` was designed to be processor that coordinates backpressure like the `Flowable.publish` operators do. It includes internal optimizations such as operator-fusion when subscribing it to the right kind of source.

Since users can retain the reference to the processor itself, they could, in concept, call the `onXXX` methods and possibly cause trouble. The same is true for the `offer` method which, when called while the aforementioned fusion is taking place, leads to undefined behavior in 2.x.

With 3.x, the `offer` method will throw an `IllegalStateException` and not disturb the internal state of the processor.

:information_source: **Further references:** PR #6799

Group abandonment in groupBy

The `groupBy` operator is one of the peculiar operators that signals reactive sources as its main output where consumers are expected to subscribe to these inner sources as well. Consequently, if the main sequence gets cancelled (i.e., the `Flowable<GroupedFlowable<T>>` itself), the consumers should still keep receiving items on their groups but no new groups should be created. The original source can then only be cancelled if all of such inner consumers have cancelled as well.

However, in 2.x, nothing is forcing the consumption of the inner sources and thus groups may be simply ignored altogether, preventing the cancellation of the original source and possibly leading to resource leaks.

With 3.x, the behavior of `groupBy` has been changed so that when it emits a group, the downstream has to subscribe to it synchronously. Otherwise, the group is considered "abandoned" and terminated. This way, abandoned groups won't prevent the cancellation of the original source. If a late consumer still subscribes to the group, the item that triggered the group creation will be still available.

Synchronous subscription means the following flow setups **will cause abandonment** and possibly group re-creation:

groupBy abandonment example

```
// observeOn creates a time gap between group emission
// and subscription
source.groupBy(v -> v)
.observeOn(Schedulers.computation())
.flatMap(g -> g)

// subscribeOn creates a time gap too
source.groupBy(v -> v)
.flatMap(g -> g.subscribeOn(Schedulers.computation()))
```

Since the groups are essentially hot sources, one should use `observeOn` to move the processing of the items safely to another thread anyway:

```
source.groupBy(v -> v)
.flatMap(g ->
    g.observeOn(Schedulers.computation())
    .map(v -> v + 1)
)
```

:information_source: **Further references:** Issue #6596, PR #6642

Backpressure in groupBy

The `Flowable.groupBy` operator is even more peculiar in a way that it has to coordinate backpressure from the consumers of its inner group and request from its original `Flowable`. The complication is, such requests can lead to a creation of a new group, a new item for the group that itself requested or a new item for a completely different group altogether. Therefore, groups can affect each other's ability to receive items and can hang the sequence, especially if some groups don't get to be consumed at all.

This latter can happen when groups are merged via `flatMap` where the number of individual groups is greater than the `flatMap`'s concurrency level (defaul 128) so fresh groups won't get subscribed to and old ones may not complete to make room. With `concatMap`, the same issue can manifest immediately.

Since RxJava is non-blocking, such silent hangs are difficult to detect and diagnose (i.e., no thread is blocking in `groupBy` or `flatMap`). Therefore, 3.x changed the behavior of `groupBy` so that if the immediate downstream is unable to receive a new group, the sequence terminates with `MissingBackpressureException`:

groupBy backpressure example

```
Flowable.range(1, 1000)
.groupBy(v -> v)
.flatMap(v -> v, 16)
.test()
.assertError(MissingBackpressureException);
```

The error message will also indicate the group index:

> *Unable to emit a new group (#16) due to lack of requests. Please make sure the downstream can always accept a new group and each group is consumed for the whole operator to be able to proceed.*

Increasing the concurrency level to the right amount (or `Integer.MAX_VALUE` if the number of groups is not known upfront) should resolve the problem:

```
.flatMap(v -> v, 1000)
```

:information_source: **Further references:** Issue #6641, PR #6740

Window abandonment in window

Similar to groupBy, the `window` operator emits inner reactive sequences that should still keep receiving items when the outer sequence is cancelled (i.e., working with only a limited set of windows). Similarly, when all window consumers cancel, the original source should be cancelled as well.

However, in 2.x, nothing is forcing the consumption of the inner sources and thus windows may be simply ignored altogether, preventing the cancellation of the original source and possibly leading to resource leaks.

With 3.x, the behavior of all `window` operators has been changed so that when it emits a group, the downstream has to subscribe to it synchronously. Otherwise, the window is considered "abandoned" and terminated. This way, abandoned windows won't prevent the cancellation of the original source. If a late consumer still subscribes to the window, the item that triggered the window creation *may be* still available.

Synchronous subscription means the following flow setups **will cause abandonment**:

window abandonment example

```
// observeOn creates a time gap between window emission
// and subscription
source.window(10, 5)
.observeOn(Schedulers.computation())
.flatMap(g -> g)

// subscribeOn creates a time gap too
source.window(1, TimeUnit.SECONDS)
.flatMap(g -> g.subscribeOn(Schedulers.computation()))
```

Since the windows are essentially hot sources, one should use `observeOn` to move the processing of the items safely to another thread anyway:

```
source.window(1, TimeUnit.SECONDS)
.flatMap(g ->
    g.observeOn(Schedulers.computation())
    .map(v -> v + 1)
)
```

:information_source: **Further references:** PR #6758, PR #6761, PR #6762

CompositeException cause generation

In 1.x and 2.x, calling the `CompositeException.getCause()` method resulted in a generation of a chain of exceptions from the internal list of aggregated exceptions. This was mainly done because Java 6 lacks the suppressed exception feature of Java 7+ exceptions. However, the implementation was possibly mutating exceptions or, sometimes, unable to establish a chain at all. Given the source of the original contribution of the method, it was risky to fix the issues with it in 2.x.

With 3.x, the method constructs a cause exception that when asked for a stacktrace, generates an output without touching the aggregated exceptions (which is IDE friendly and should be navigable):

stacktrace example

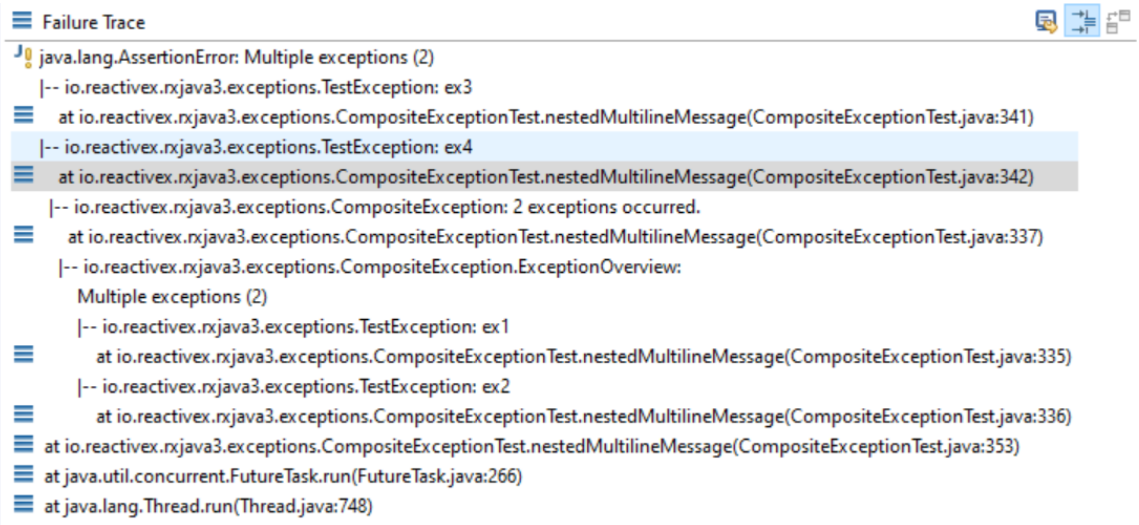
```
Multiple exceptions (2)
|-- io.reactivex.rxjava3.exceptions.TestException: ex3
    at
io.reactivex.rxjava3.exceptions.CompositeExceptionTest.nestedMultilineMessage(CompositeExceptionTest.java:341)

|-- io.reactivex.rxjava3.exceptions.TestException: ex4
    at
io.reactivex.rxjava3.exceptions.CompositeExceptionTest.nestedMultilineMessage(CompositeExceptionTest.java:342)

  |-- io.reactivex.rxjava3.exceptions.CompositeException: 2 exceptions occurred.
      at
io.reactivex.rxjava3.exceptions.CompositeExceptionTest.nestedMultilineMessage(CompositeExceptionTest.java:337)

    |-- io.reactivex.rxjava3.exceptions.CompositeException.ExceptionOverview:
        Multiple exceptions (2)
        |-- io.reactivex.rxjava3.exceptions.TestException: ex1
            at
io.reactivex.rxjava3.exceptions.CompositeExceptionTest.nestedMultilineMessage(CompositeExceptionTest.java:335)

        |-- io.reactivex.rxjava3.exceptions.TestException: ex2
            at
io.reactivex.rxjava3.exceptions.CompositeExceptionTest.nestedMultilineMessage(CompositeExceptionTest.java:336)
```



```
java.lang.AssertionError: Multiple exceptions (2)
  |-- io.reactivex.rxjava3.exceptions.TestException: ex3
    at io.reactivex.rxjava3.exceptions.CompositeExceptionTest.nestedMultilineMessage(CompositeExceptionTest.java:341)
  |-- io.reactivex.rxjava3.exceptions.TestException: ex4
    at io.reactivex.rxjava3.exceptions.CompositeExceptionTest.nestedMultilineMessage(CompositeExceptionTest.java:342)
    |-- io.reactivex.rxjava3.exceptions.CompositeException: 2 exceptions occurred.
      at io.reactivex.rxjava3.exceptions.CompositeExceptionTest.nestedMultilineMessage(CompositeExceptionTest.java:337)
      |-- io.reactivex.rxjava3.exceptions.CompositeException.ExceptionOverview:
        Multiple exceptions (2)
        |-- io.reactivex.rxjava3.exceptions.TestException: ex1
          at io.reactivex.rxjava3.exceptions.CompositeExceptionTest.nestedMultilineMessage(CompositeExceptionTest.java:335)
        |-- io.reactivex.rxjava3.exceptions.TestException: ex2
          at io.reactivex.rxjava3.exceptions.CompositeExceptionTest.nestedMultilineMessage(CompositeExceptionTest.java:336)
at io.reactivex.rxjava3.exceptions.CompositeExceptionTest.nestedMultilineMessage(CompositeExceptionTest.java:353)
at java.util.concurrent.FutureTask.run(FutureTask.java:266)
at java.lang.Thread.run(Thread.java:748)
```

:information_source: **Further references:** Issue #6747, PR #6748

Parameter validation exception change

Some standard operators in 2.x throw `IndexOutOfBoundsException` when the respective argument was invalid. For consistency with other parameter validation exceptions, the following operators now throw `IllegalArgumentException` instead:

- `skip`
- `skipLast`
- `takeLast`
- `takeLastTimed`

:information_source: **Further references:** PR #6831, PR #6835

From-callbacks upfront cancellation

In 2.x, canceling sequences created via `fromRunnable` and `fromAction` were inconsistent with other `fromX` sequences when the downstream cancelled/disposed the sequence immediately.

In 3.x, such upfront cancellation will not execute the given callback.

from callback example

```
Runnable run = mock(Runnable.class);

Completable.fromRunnable(run)
.test(true); // cancel upfront

verify(run, never()).run();
```

:information_source: **Further references:** PR #6873

Using cleanup order

The operator `using` has an `eager` parameter to determine when the resource should be cleaned up: `true` means before-termination and `false` means after-termination. Unfortunately, this setting didn't affect the cleanup order upon donwstream cancellation and was always cleaning up the resource before canceling the upstream.

In 3.x, the cleanup order is now consistent when the sequence terminates or gets cancelled: `true` means before-termination or before canceling the upstream, `false` means after-termination or after canceling the upstream.

:information_source: **Further references:** Issue #6347, PR #6534

API changes

A major release allows cleaning up and improving the API surface by adding, changing or removing elements all across. With 3.x, there are several of such changes that require some explanations.

Functional interfaces

RxJava 2.x introduced a custom set of functional interfaces in `io.reactivex.functions` so that the use of the library is possible with the same types on Java 6 and Java 8. A secondary reason for such custom types is that the standard Java 8 function types do not support throwing any checked exceptions, which in itself can result in some inconvenience when using RxJava operators.

Despite RxJava 3 being based on Java 8, the issues with the standard Java 8 functional interfaces persist, now with possible desugaring issues on Android and their inability to throw checked exceptions. Therefore, 3.x kept the custom interfaces, but the `@FunctionalInterface` annotation has been applied to them (which is safe/ignored on Android).

```
@FunctionalInterface
interface Function<@NonNull T, @NonNull R> {
    R apply(T t) throws Throwable;
}
```

In addition, Java 8 allows declaring annotations on type argument and type argument use individually and thus all functional interfaces have received nullability annotations.

:information_source: **Further references:** PR #6840, PR #6791, PR #6795

Wider throws

One small drawback with the custom `throws Exception` in the functional interfaces is that some 3rd party APIs may throw a checked exception that is not a descendant of `Exception`, or simply throw `Throwable`.

Therefore, with 3.x, the functional interfaces as well as other support interfaces have been widened and declared with `throws Throwable` in their signature.

This widening should be inconsequential for lambda-based or class-implementations provided to the RxJava methods:

```
source.map(v -> {
    if (v == 0) {
        throw new Exception();
    }
    return v;
});

source.filter(new Predicate<Integer>() {
    @Override
    public boolean test() throws Exception {
        throw new IOException();
    }
});
```

I.e., there is no need to change `throws Exception` to `throws Throwable` just for the sake of it.

However, if one uses these functional interfaces outside:

```
static void Integer callFunction(
        Function<Integer, Integer> function, Integer value) throws Exception {
```

```
    return function.apply(value);
}
```

the widening of `throws` will have to be propagated:

```
static void Integer callFunction(
        Function<Integer, Integer> function, Integer value) throws Throwable {
    return function.apply(value);
}
```

:information_source: **Further references:** PR #6511, PR #6579

New Types

Supplier

RxJava 2.x already supported the standard `java.util.concurrent.Callable` whose `call` method is declared with `throws Exception` by default. Unfortunately, when our custom functional interfaces were widened to `throws Throwable`, it was impossible to widen `Callable` because in Java, implementations can't widen the `throws` clause, only narrow or abandon it.

Therefore, 3.x introduces the `io.reactivex.rxjava3.functions.Supplier` interface that defines the widest `throws` possible:

```
interface Supplier<@NonNull R> {
    R get() throws Throwable;
}
```

⚠ Note on running "Organize Imports" from the IDE

Due to naming matches, IDE's tend to import `java.util.function.Supplier` instead of picking RxJava's `io.reactivex.rxjava3.functions.Supplier`. Also IDEs tend to give non-descriptive errors such as "Suppliercan't be converted to Supplier", omitting the fact about the package differences.

⚠ Signature change

To comply with the support for wider throws functional interfaces, many operators used to take `java.util.concurrent.Callable` now take `io.reactivex.rxjava3.functions.Supplier` instead. If the operator was used with a lambda, only recompilation is needed:

```
Flowable.defer(() -> Flowable.just(Math.random()));
```

However, if explicit implementation was used:

```
Flowable.defer(new Callable<Double>() {
    @Override
    public Double call() throws Exception {
        return Math.random();
    }
});
```

the **interface type** (`Callable` -> `Supplier`) and the **method name** (`call` -> `get`) has to be adjusted:

```
Flowable.defer(new Supplier<Double>() {
    @Override
    public Double get() throws Exception {
        return Math.random();
    }
});
```

See the API signature changes section on which operators are affected.

:information_source: **Further references:** PR #6511

Converters

In 2.x, the `to()` operator used the generic `Function` to allow assembly-time conversion of flows into arbitrary types. The drawback of this approach was that each base reactive type had the same `Function` interface in their method signature, thus it was impossible to implement multiple converters for different reactive types within the same class. To work around this issue, the `as` operator and `XConverter` interfaces have been introduced in 2.x, which interfaces are distinct and can be implemented on the same class. Changing the signature of `to` in 2.x was not possible due to the pledged binary compatibility of the library.

From 3.x, the `as()` methods have been removed and the `to()` methods now each work with their respective `XConverter` interfaces (hosted in package `io.reactivex.rxjava3.core`):

- `Flowable.to(Function<Flowable<T>, R>)` -> `Flowable.to(FlowableConverter<T, R>)`
- `Observable.to(Function<Observable<T>, R>)` -> `Observable.to(ObservableConverter<T, R>)`
- `Maybe.to(Function<Flowable<T>, R>)` -> `Maybe.to(MaybeConverter<T, R>)`
- `Single.to(Function<Flowable<T>, R>)` -> `Maybe.to(SingleConverter<T, R>)`
- `Completable.to(Function<Completable, R>)` -> `Completable.to(CompletableConverter<R>)`
- `ParallelFlowable.to(Function<ParallelFlowable<T>, R)` -> `ParallelFlowable.to(ParallelFlowableConverter<T, R>)`

If one was using these methods with a lambda expression, only a recompilation is needed:

```
// before
source.to(flowable -> flowable.blockingFirst());

// after
source.to(flowable -> flowable.blockingFirst());
```

If one was implementing a Function interface (typically anonymously), the interface type, type arguments and the `throws` clause have to be adjusted

```
// before
source.to(new Function<Flowable<Integer>, Integer>() {
    @Override
    public Integer apply(Flowable<Integer> t) throws Exception {
        return t.blockingFirst();
    }
});

// after
source.to(new FlowableConverter<Integer, Integer>() {
    @Override
    public Integer apply(Flowable<Integer> t) {
        return t.blockingFirst();
    }
});
```

:information_source: **Further references:** Issue #5654, PR #6514

Moved components

Disposables

Moving to Java 8 and Android's desugaring tooling allows the use of static interface methods instead of separate factory classes. The support class `io.reactivex.disposables.Disposables` was a prime candidate for moving all of its methods into the `Disposable` interface itself (`io.reactivex.rxjava3.disposables.Disposable`).

Uses of the factory methods:

```
Disposable d = Disposables.empty();
```

should now be turned into:

```
Disposable d = Disposable.empty();
```

:information_source: **Further references:** PR #6781

DisposableContainer

Internally, RxJava 2.x uses an abstraction of a disposable container instead of using `CompositeDisposable` everywhere, allowing a more appropriate container type to be used. This is achieved via an internal `DisposableContainer` implemented by `CompositeDisposable` as well as other internal components. Unfortunately, since the public class referenced an internal interface, RxJava was causing warnings in OSGi environments.

In RxJava 3, the `DisposableContainer` is now part of the public API under `io.reactivex.rxjava3.disposables.DisposableContainer` and no longer causes OSGi issues.

:information_source: **Further references:** Issue #6742, PR #6745

API promotions

The RxJava 2.2.x line has still a couple of **experimental** operators (but no **beta**) operators, which have been promoted to standard with 3.x:

Flowable promotions

- `dematerialize(Function)`

Observable promotions

- `dematerialize(Function)`

Maybe promotions

- `doOnTerminate(Action)`
- `materialize()`

Single promotions

- `dematerialize(Function)`
- `materialize()`

Completable promotions

- `delaySubscription(long, TimeUnit)`
- `delaySubscription(long, TimeUnit, Scheduler)`
- `materialize()`

:information_source: **Further references:** PR #6537

API additions

RxJava 3 received a considerable amount of new operators and methods across its API surface. Brand new operators introduced are marked with

🟡 in their respective **Available in:** listings

replay with eagerTruncate

Available in: ✅ `Flowable` , ✅ `Observable` , ○ `Maybe` , ○ `Single` , ○ `Completable`

A limitation with the bounded `replay` operator is that to allow continuous item delivery to slow consumers, a linked list of the cached items has to be maintained. By default, the head node of this list is moved forward when the boundary condition (size, time) mandates it. This setup avoids allocation in exchange for retaining one "invisible" item in the linked list. However, sometimes this retention is unwanted and the allocation overhead of a clean node is acceptable. In 2.x, the `ReplaySubject` and `ReplayProcessor` implementations already allowed for such behavior, but the instance `replay()` operators did not.

With 3.x, the `replay` operators (both connectable and multicasting variants) received overloads, defining an `eagerTruncate` option that performs this type of head node cleanup.

Flowable

- `replay(int, boolean)`
- `replay(long, TimeUnit, Scheduler, boolean)`
- `replay(int, long, TimeUnit, Scheduler, boolean)`
- `replay(Function, int, boolean)`

- `replay(Function, long, TimeUnit, Scheduler, boolean)`
- `replay(Function, int, long, TimeUnit, Scheduler, boolean)`

Observable

- `replay(int, boolean)`
- `replay(long, TimeUnit, Scheduler, boolean)`
- `replay(int, long, TimeUnit, Scheduler, boolean)`
- `replay(Function, int, boolean)`
- `replay(Function, long, TimeUnit, Scheduler, boolean)`
- `replay(Function, int, long, TimeUnit, Scheduler, boolean)`

:information_source: **Further references:** Issue #6475, PR #6532

concatMap with Scheduler

Available in: ✅ `Flowable`, ✅ `Observable`, ○ `Maybe`, ○ `Single`, ○ `Completable`

A property of the `concatMap` operator is that the `mapper` function may be invoked either on the subscriber's thread or the currently completing inner source's thread. There is no good way to control this thread of invocation from the outside, therefore, new overloads have been added in 3.x with an additional `Scheduler` parameter:

Flowable

- `concatMap(Function, int, Scheduler)`
- `concatMapDelayError(Function, int, boolean, Scheduler)`

Observable

- `concatMap(Function, int, Scheduler)`
- `concatMapDelayError(Function, int, boolean, Scheduler)`

:information_source: **Further references:** Issue #6447, PR #6538

Schedulers.from fair mode

By default, `Schedulers.from` executes work on the supplied `Executor` in an eager mode, running as many tasks as available. This can cause some unwanted lack of interleaving between these tasks and external tasks submitted to the same executor. To remedy the situation, a new mode and overload has been added so that the `Scheduler` returned by `Schedulers.from` runs tasks one by one, allowing other external tasks to be interleaved.

- `Schedulers.from(Executor, boolean, boolean)`

:information_source: **Further references:** Issue #6696, Issue #6697, PR #6744

blockingForEach with buffer size

Available in: ✅ `Flowable`, ✅ `Observable`, ○ `Maybe`, ○ `Single`, ○ `Completable`

The underlying `blockingIterable` operator had already the option to specify the internal buffer size (and prefetch amounts), which is now exposed via new `blockingForEach` overloads

- `Flowable.blockingForEach(Consumer, int)`
- `Observable.blockingForEach(Consumer, int)`

:information_source: **Further references:** Issue #6784, PR #6800

blockingSubscribe

Available in: ✅ `Flowable`, ✅ `Observable`, 🟡 `Maybe`, 🟡 `Single`, 🟡 `Completable`

For API consistency, the callback-based `blockingSubscribe` methods have been introduced to `Maybe`, `Single` and `Completable` respectively.

Maybe

- `blockingSubscribe()`
- `blockingSubscribe(Consumer)`
- `blockingSubscribe(Consumer, Consumer)`
- `blockingSubscribe(Consumer, Consumer, Action)`
- `blockingSubscribe(MaybeObserver)`

Single

- `blockingSubscribe()`
- `blockingSubscribe(Consumer)`
- `blockingSubscribe(Consumer, Consumer)`
- `blockingSubscribe(SingleObserver)`

Completable

- `blockingSubscribe()`
- `blockingSubscribe(Action)`
- `blockingSubscribe(Action, Consumer)`
- `blockingSubscribe(CompletableObserver)`

:information_source: **Further references:** Issue #6852, PR #6862

Maybe.delay with delayError

Available in: ✅ `Flowable`, ✅ `Observable`, ✅ `Maybe`, ✅ `Single`, ✅ `Completable`

The option, available in every other reactive type, to delay the errors optionally as well was missing from `Maybe`.

- `delay(long, TimeUnit, boolean)`
- `delay(long, TimeUnit, Scheduler, boolean)`

:information_source: **Further references:** Issue #6863, PR #6864

onErrorComplete

Available in: 🟡 `Flowable`, 🟡 `Observable`, ✅ `Maybe`, 🟡 `Single`, ✅ `Completable`

Upon an error, the sequence is completed (conditionally) instead of signaling the error.

Flowable

- `onErrorComplete()`
- `onErrorComplete()`

Observable

- `onErrorComplete()`
- `onErrorComplete()`

Single

- `onErrorComplete()`
- `onErrorComplete()`

:information_source: **Further references:** Issue #6852, PR #6867

Completable.onErrorResumeWith

Available in: ✅ `Flowable`, ✅ `Observable`, ✅ `Maybe`, ✅ `Single`, 🟡 `Completable`

This operator (under the name `onErrorResumeNext` now renamed) was already available everywhere else and was accidentally left out of `Completable`.

- `onErrorResumeWith(Completable)`

:information_source: **Further references:** Issue #6852, PR #6868

retryUntil

Available in: ✅ `Flowable`, ✅ `Observable`, ✅ `Maybe`, 🟡 `Single`, 🟡 `Completable`

The operator was missing from `Single` and `Completable`.

- `Single.retryUntil(BooleanSupplier)`
- `Completable.retryUntil(BooleanSupplier)`

:information_source: **Further references:** Issue #6852, PR #6869

switchOnNext

Available in: ✅ `Flowable`, ✅ `Observable`, 🟡 `Maybe`, 🟡 `Single`, 🟡 `Completable`

Added the static version of the `switchMap` operator, `switchOnNext` and `switchOnNextDelayError`, to `Maybe`, `Single` and `Completable`.

- `Maybe.switchOnNext(Function)`
- `Single.switchOnNext(Function)`
- `Completable.switchOnNext(Function)`
- `Maybe.switchOnNextDelayError(Function)`
- `Single.switchOnNextDelayError(Function)`
- `Completable.switchOnNextDelayError(Function)`

:information_source: **Further references:** Issue #6852, PR #6870

Maybe.dematerialize

Available in: ✅ `Flowable`, ✅ `Observable`, 🟡 `Maybe`, ✅ `Single`, ✅ `Completable`

The operator was already added to the other reactive types before.

- `dematerialize(Function)`

:information_source: **Further references:** Issue #6852, PR #6871

from conversions

Several operators have been added across:

Operator	F	O	M	S	C
`fromAction`	🟡	🟡	✅	○ (23)	✅
`fromCompletable`	🟡	🟡	✅	○ (72)	○ (73)
`fromMaybe`	🟡	🟡	○ (73)	🟡	✅
`fromObservable`	🟡	○ (73)	🟡	✅	✅
`fromPublisher`	✅	✅	🟡	✅	✅
`fromRunnable`	🟡	🟡	✅	○ (23)	✅

`fromSingle`	🟡	🟡	✅	○ (73)	✅

Flowable

- `fromAction(Action)`
- `fromCompletable(CompletableSource)`
- `fromMaybe(MaybeSource)`
- `fromObservable(ObservableSource, BackpressureStrategy)`
- `fromRunnable(Runnable)`
- `fromSingle(Runnable)`

Observable

- `fromAction(Action)`
- `fromCompletable(CompletableSource)`
- `fromMaybe(MaybeSource)`
- `fromRunnable(Runnable)`
- `fromSingle(Runnable)`

Maybe

- `fromObservable(ObservableSource)`
- `fromPublisher(Publisher)`

Single

- `fromMaybe(ObservableSource)`

:information_source: **Further references:** Issue #6852, PR #6873

timestamp and timeInterval

Available in: ✅ `Flowable`, ✅ `Observable`, 🟡 `Maybe`, 🟡 `Single`, ○ `Completable`

These operators were already available for `Flowable` and `Observable`, now added to `Single` and `Maybe`.

Maybe

- `timeInterval()`
- `timeInterval(TimeUnit)`
- `timeInterval(Scheduler)`
- `timeInterval(TimeUnit, Scheduler)`
- `timestamp()`
- `timestamp(TimeUnit)`
- `timestamp(Scheduler)`
- `timestamp(TimeUnit, Scheduler)`

Single

- `timeInterval()`
- `timeInterval(TimeUnit)`
- `timeInterval(Scheduler)`
- `timeInterval(TimeUnit, Scheduler)`
- `timestamp()`
- `timestamp(TimeUnit)`
- `timestamp(Scheduler)`
- `timestamp(TimeUnit, Scheduler)`

:information_source: **Further references:** Issue #6852, PR #6874

toFuture

Available in: ✅ `Flowable`, ✅ `Observable`, 🟡 `Maybe`, ✅ `Single`, 🟡 `Completable`

This operator was already available elsewhere, now added to `Maybe` and `Completable`.

- `Maybe.toFuture()`
- `Completable.toFuture()`

:information_source: **Further references:** Issue #6852, PR #6875

ofType

Available in: ✅ `Flowable`, ✅ `Observable`, 🟡 `Maybe`, 🟡 `Single`, ○ `Completable`

This operator was already available in `Flowable` and `Observable`, now added to `Maybe` and `Single`.

- `Maybe.ofType(Class)`
- `Single.ofType(Class)`

:information_source: **Further references:** Issue #6852, PR #6876

doOnLifecycle

Available in: ✅ `Flowable`, ✅ `Observable`, 🟡 `Maybe`, 🟡 `Single`, 🟡 `Completable`

This operator was already available in `Flowable` and `Observable`, now added to `Maybe`, `Single` and `Completable`.

- `Maybe.doOnLifecycle()`
- `Single.doOnLifecycle()`
- `Completable.doOnLifecycle()`

:information_source: **Further references:** Issue #6852, PR #6877

concatMap to another type

Available in: ✅ `Flowable`, ✅ `Observable`, 🟡 `Maybe`, 🟡 `Single`, ○ `Completable`

Added varios concatMap-based transformations between `Maybe`, `Single` and `Completable` for `Maybe` and `Single`. These are essentially aliases to the respective `flatMap` operators for better discoverability.

- `Maybe.concatMapCompletable(Function)`
- `Maybe.concatMapSingle(Function)`
- `Single.concatMapCompletable(Function)`
- `Single.concatMapMaybe(Function)`
- `Single.concatMap(Function)`

:information_source: **Further references:** Issue #6852, PR #6879

concat with delayError

Available in: ✅ `Flowable`, ✅ `Observable`, 🟡 `Maybe`, 🟡 `Single`, 🟡 `Completable`

The delayError variants of the `concat` operator were missing across.

Maybe

- `Maybe.concatArrayEagerDelayError(Maybe...)`
- `Maybe.concatDelayError(Publisher, int)`

Single

- `Single.concatArrayDelayError(Single...)`
- `Single.concatArrayEagerDelayError`
- `Single.concatDelayError(Iterable)`
- `Single.concatDelayError(Publisher)`
- `Single.concatDelayError(Publisher, int)`

Completable

- `Completable.concatArrayDelayError(Completable...)`
- `Completable.concatDelayError(Iterable)`
- `Completable.concatDelayError(Publisher)`
- `Completable.concatDelayError(Publisher, int)`

:information_source: **Further references:** Issue #6852, PR #6881

Single.mergeArray

Available in: ✅ `Flowable`, ✅ `Observable`, ✅ `Maybe`, 🟡 `Single`, ✅ `Completable`

The operator was already available elsewhere, now added to `Single`.

- `mergeArray(SingleSource...)`
- `mergeArrayDelayError(SingleSource...)`

:information_source: **Further references:** Issue #6852, PR #6882

Completable.sequenceEqual

Available in: ✅ `Flowable`, ✅ `Observable`, ✅ `Maybe`, ✅ `Single`, 🟡 `Completable`

The operator was already available elsewhere, now added to `Completable`.

- `sequenceEqual`

:information_source: **Further references:** Issue #6852, PR #6884

startWith

Available in:

source \ other	F	O	M	S	C
Flowable	✅	⚪	🟡	🟡	🟡
Observable	⚪	✅	🟡	🟡	🟡
Maybe	🟡	🟡	🟡	🟡	🟡
Single	🟡	🟡	🟡	🟡	🟡
Completable	✅	✅	🟡	🟡	✅

Added overloads for better continuation support between the reactive types.

Flowable

- `startWith(MaybeSource)`
- `startWith(SingleSource)`
- `startWith(CompletableSource)`

Observable

- `startWith(MaybeSource)`
- `startWith(SingleSource)`

- `startWith(CompletableSource)`

Maybe

- `startWith(Publisher)`
- `startWith(ObservableSource)`
- `startWith(MaybeSource)`
- `startWith(SingleSource)`
- `startWith(CompletableSource)`

Single

- `startWith(Publisher)`
- `startWith(ObservableSource)`
- `startWith(MaybeSource)`
- `startWith(SingleSource)`
- `startWith(CompletableSource)`

Completable

- `startWith(MaybeSource)`
- `startWith(SingleSource)`

:information_source: **Further references:** Issue #6852, PR #6885

Completable.onErrorReturn

Available in: ✅ `Flowable`, ✅ `Observable`, ✅ `Maybe`, ✅ `Single`, 🟡 `Completable`

The operators `onErrorReturn` and `onErrorReturnItem` weres available everywhere else and are now added to `Completable`.

- `onErrorReturn`
- `onErrorReturnItem`

:information_source: **Further references:** Issue #6852, PR #6886

safeSubscribe

Available in: ✅ `Flowable`, ✅ `Observable`, 🟡 `Maybe`, 🟡 `Single`, 🟡 `Completable`

The method was already available in `Flowable` and `Observable`, now added to `Maybe`, `Single` and `Completable` for API consistency.

- `Maybe.safeSubscribe(MaybeObserver)`
- `Single.safeSubscribe(SingleObserver)`
- `Completable.safeSubscribe(CompletableObserver)`

:information_source: **Further references:** Issue #6852, PR #6887

Single.flatMap

Available in: ✅ `Flowable`, ✅ `Observable`, ✅ `Maybe`, 🟡 `Single`, ✅ `Completable`

Add two overloads of `flatMap` to `Single`: one to transform the success or error signals into a new `SingleSource`, one to combine the original success value with the success value of the inner sources:

- `flatMap(Function, Function)`
- `flatMap(Function, BiFunction)`

:information_source: **Further references:** Issue #6852, PR #6893

concatEager and concatEagerDelayError

Available in: 🟡 `Flowable`, 🟡 `Observable`, 🟡 `Maybe`, 🟡 `Single`, ⚪ `Completable`

Add various `concatEager` and `concatEagerDelayError` overloads across the reactive types.

Flowable

- `concatEagerDelayError(Iterable)`
- `concatEagerDelayError(Iterable, int, int)`
- `concatEagerDelayError(Publisher)`
- `concatEagerDelayError(Publisher, int, int)`

Observable

- `concatEagerDelayError(Iterable)`
- `concatEagerDelayError(Iterable, int, int)`
- `concatEagerDelayError(Publisher)`
- `concatEagerDelayError(Publisher, int, int)`

Maybe

- `concatEager(Iterable, int)`
- `concatEager(Publisher, int)`
- `concatEagerDelayError(Iterable)`
- `concatEagerDelayError(Iterable, int)`
- `concatEagerDelayError(Publisher)`
- `concatEagerDelayError(Publisher, int)`

Single

- `concatEager(Iterable, int)`
- `concatEager(Publisher, int)`
- `concatEagerDelayError(Iterable)`
- `concatEagerDelayError(Iterable, int)`
- `concatEagerDelayError(Publisher)`
- `concatEagerDelayError(Publisher, int)`

:information_source: **Further references:** Issue #6880, PR #6899

fromSupplier

Available in: 🟡 `Flowable`, 🟡 `Observable`, 🟡 `Maybe`, 🟡 `Single`, 🟡 `Completable`

With the new type `io.reactivex.rxjava3.functions.Supplier` comes a new wrapper operator `fromSupplier` to complement `fromCallable` in all the reactive types.

- `Flowable.fromSupplier`
- `Observable.fromSupplier`
- `Maybe.fromSupplier`
- `Single.fromSupplier`
- `Completable.fromSupplier`

:information_source: **Further references:** PR #6529

ParallelFlowable.flatMapIterable

Available in: ✅ `Flowable`, ✅ `Observable`, ○ `Maybe`, ○ `Single`, ○ `Completable`

🟡 `ParallelFlowable`

The operator was already available in `Flowable` and `Observable`, now added to `ParallelFlowable`.

- `flatMapIterable(Function)`
- `flatMapIterable(Function, int)`

flatMapIterable example

```
Flowable.range(1, 10)
.parallel()
.runOn(Schedulers.computation())
.flatMapIterable(v -> Arrays.asList(v, v + 1));
```

:information_source: **Further references:** PR #6798

Java 8 additions

Now that the API baseline is set to Java 8, RxJava can now support the new types of Java 8 directly, without the need of an external library (such as the RxJavaJdk8Interop library, now discontinued).

:warning: Note that the Android desugar may not support all Java 8 APIs for all target possible Android API levels, however, their mere existence in the RxJava class files should not cause any trouble.

Java 8 functional interfaces

:warning: RxJava 3 doesn't support working with Java 8 functional interfaces directly because it prefers its own custom set of functional interfaces with a wider range of platform support and exception handling.

However, Java 8 has language support for a relatively convenient way to convert a Java 8 functional interface to its RxJava 3 equivalent via method handles:

```
java.util.function.Function<Integer, Integer> f = a -> a + 1;

Flowable.range(1, 10)
.map(f::apply)
;

// or
io.reactivex.rxjava3.functions.Function<Integer, Integer> f2 = f::apply;
```

Unfortunately, the reverse direction is not possible because Java 8's functional interfaces do not allow throwing a checked exception.

In general, the distinction between the two sets of interfaces shouldn't be a practical problem because unlike Java 8's `java.util.Collectors`, there is no repository of pre-made functional interface implementations out there that would require more direct support from RxJava.

fromOptional

Available in: ● `Flowable`, ● `Observable`, ● `Maybe`, ○ `Single`, ○ `Completable`

Given an existing, constant reference of a `java.util.Optional`, turn it into a `Flowable`, `Observable` or `Maybe` source, emitting its value immediately or completing immediately.

:bulb: There is no `Single` variant because it *has to be non-empty*. No `Completable` either because *it is always empty*.

- `Flowable.fromOptional`
- `Observable.fromOptional`
- `Maybe.fromOptional`

fromOptional example

```
Flowable<Integer> zero = Flowable.fromOptional(Optional.empty());

Observable<Integer> one = Flowable.fromOptional(Optional.of(1));

Maybe<Integer> maybe = Flowable.fromOptional(Optional.ofNullable(valueMaybeNull));
```

:information_source: **Further references:** Issue #6776, PR #6765, PR #6783, PR #6797

fromStream

Available in: ● `Flowable`, ● `Observable`, ○ `Maybe`, ○ `Single`, ○ `Completable`

Turns a `java.util.stream.Stream` into a `Flowable` or `Observable` and emits its items to consumers.

:bulb: Because `Stream` is assumed to be having zero to N items (possibly infinite), there is no good way to expose it as `Maybe`, `Single` or `Completable`.

:warning: RxJava 3 doesn't accept the primitive `Stream` types (such as `IntStream` and `DoubleStream`). These streams have to be converted into their boxed variants via their `boxed()` method. Since RxJava uses reference types, there is no way to optimize the interoperation with the primitive streams.

- `Flowable.fromStream`
- `Observable.fromStream`

fromStream example

```
Flowable<Integer> stream = Flowable.fromStream(IntStream.range(1, 10).boxed());

Observable<Integer> stream2 = Observable.fromStream(list.stream());
```

:information_source: **Further references:** Issue #6776, PR #6765, PR #6797

fromCompletionStage

Available in: ● `Flowable`, ● `Observable`, ● `Maybe`, ● `Single`, ● `Completable`

Wrap a `java.util.concurrent.CompletionStage` instance (such as `CompletableFuture`) into a reactive type and expose its single value or failure as the appropriate reactive signal.

:bulb: A `CompletionStage` is like a hot source that is already executing or has already terminated, thus the wrapper is only there to observe the outcome, not to initiate the computation the stage represents.

:warning: Note that the standard Java 8 `CompletionStage` interface doesn't support cancellation, thus canceling the reactive flow will not stop the given `CompletionStage`.

- `Flowable.fromCompletionStage`
- `Observable.fromCompletionStage`
- `Maybe.fromCompletionStage`
- `Single.fromCompletionStage`
- `Completable.fromCompletionStage`

fromCompletionStage example

```
Flowable<Integer> someAsync = Flowable.fromCompletionStage(
    operation.getAsync()
);

Obervable<Integer> otherAsync = Observable.fromCompletionStage(
    CompletableFuture.completedFuture(1)
);

Maybe<Object> failed = Maybe.fromCompletionStage(
    CompletableFurure.completedFuture(0)
    .thenAccept(v -> { throw new RuntimeException(); })
```

:information_source: **Further references:** Issue #6776, PR #6765, PR #6783, PR #6797

mapOptional

Available in: ● `Flowable`, ● `Observable`, ● `Maybe`, ● `Single`, ○ `Completable`

● `ParallelFlowable`

Transform the upstream item(s) into `java.util.Optional` instances, then emit the non-empty value, or if the `Optional` is empty, skip to the next upstream value.

:bulb: `Completable` has no items to map.

- `Flowable.mapOptional`
- `Observable.mapOptional`
- `Maybe.mapOptional`
- `Single.mapOptional`
- `ParallelFlowable.mapOptional(Function)`
- `ParallelFlowable.mapOptional(Function, BiFunction)`
- `ParallelFlowable.mapOptional(Function, ParallelFailureHandling)`

mapOptional example

```
Flowable.range(1, 10)
.mapOptional(v -> v % 2 == 0 ? Optional.of(v) : Optional.empty());

Flowable.range(1, 10)
.parallel()
.runOn(Schedulers.computation())
.mapOptional(v -> v % 2 == 0 ? Optional.of(v) : Optional.empty());
.sequential();
```

:information_source: **Further references:** Issue #6776, PR #6775, PR #6783, PR #6797, PR #6798

collect with Collector

Available in: ● `Flowable`, ● `Observable`, ○ `Maybe`, ○ `Single`, ○ `Completable`

● `ParallelFlowable`

Provides the ability to aggregate items of a `Flowable` or `Observable` with the help of Java 8's rich set of `Collector` implementations. See `Collectors` for its capabilities.

:bulb: Because a sequence is assumed to be having zero to N items (possibly infinite), there is no good reason to collect a `Maybe`, `Single` or `Completable`.

- `Flowable.collect(Collector)`
- `Observable.collect(Collector)`
- `ParallelFlowable.collect(Collector)`

collect example

```
Single<List<Integer>> list = Flowable.range(1, 10)
.collect(Collectors.toList());

Flowable<List<Integr>> list2 = Flowable.range(1, 10)
.parallel()
.runOn(Schedulers.computation())
.collect(Collectors.toList());
```

:information_source: **Further references:** Issue #6776, PR #6775, PR #6797, PR #6798

firstStage, singleStage, lastStage

Available in: ● `Flowable`, ● `Observable`, ○ `Maybe`, ○ `Single`, ○ `Completable`

Expose the first, only or very last item of a `Flowable` or `Observable` as a `java.util.concurrent.CompletionStage`.

:bulb: For `Maybe`, `Single` and `Completable`, the equivalent operator is called [`toCompletionStage`].

:bulb: Since a sequence can be empty, there are two variants to these methods: one that takes a default value for such an empty source and one that signals a `NoSuchElementException` via the returned `CompletionStage`. These latter methods have the [OrError] in their method name.

- [`Flowable.firstStage(T)`]
- [`Flowable.singleStage(T)`]
- [`Flowable.lastStage(T)`]
- [`Observable.firstStage(T)`]
- [`Observable.singleStage(T)`]
- [`Observable.lastStage(T)`]

stage examples

```
// Signals 1
CompletionStage<Integer> cs = Flowable.range(1, 10)
    .firstStage(11);

// Signals IndexOutOfBoundsException as the source has too many items
CompletionStage<Integer> cs1 = Flowable.just(1, 2)
    .singleStage(11);

// Signals 11
CompletionStage<Integer> cs2 = Observable.<Integer>empty()
    .lastStage(11);
```

:information_source: **Further references:** Issue [#6776], PR [#6775], PR [#6797]

firstOrErrorStage, singleOrErrorStage, lastOrErrorStage

Available in: ● `Flowable`, ● `Observable`, ○ `Maybe`, ○ `Single`, ○ `Completable`

Expose the first, only or very last item of a `Flowable` or `Observable` as a `java.util.concurrent.CompletionStage`, or signal `NoSuchElementException` of the source sequence is empty..

:bulb: For `Maybe`, `Single` and `Completable`, the equivalent operator is called [`toCompletionStage`].

:bulb: Since a sequence can be empty, there are two variants to these methods: one that takes a default value for such an empty source and one that signals a `NoSuchElementException` via the returned `CompletionStage`. The [former] do not have the `OrError` in their method name.

- [`Flowable.firstStage()`]
- [`Flowable.singleStage()`]
- [`Flowable.lastStage()`]
- [`Observable.firstStage()`]
- [`Observable.singleStage()`]
- [`Observable.lastStage()`]

stage examples

```
// Signals 1
CompletionStage<Integer> cs = Flowable.range(1, 10)
    .firstOrErrorStage();

// Signals IndexOutOfBoundsException as the source has too many items
CompletionStage<Integer> cs1 = Flowable.just(1, 2)
    .singleOrErrorStage();
```

```
// Signals NoSuchElementException
CompletionStage<Integer> cs2 = Observable.<Integer>empty()
    .lastOrErrorStage();
```

:information_source: **Further references:** Issue #6776, PR #6775, PR #6797

toCompletionStage

Available in: ○ `Flowable`, ○ `Observable`, ● `Maybe`, ● `Single`, ● `Completable`

Expose the sigle value or termination of a `Maybe`, `Single` or `Completable` as a `java.util.concurrent.CompletionStage`.

:bulb: The equivalent operators in `Flowable` and `Observable` are called `firstStage`, `singleStage`, `lastStage`, `firstOrErrorStage`, `singleOrErrorStage` and `lastOrErrorStage`.

:bulb: The `Maybe` and `Completable` operators allow defining a default completion value in case the source turns out to be empty.

- `Maybe.toCompletionStage()`
- `Maybe.toCompletionStage()`
- `Single.toCompletionStage()`
- `Completable.toCompletionStage()`

toCompletionStage example

```
// Signals 1
CompletionStage<Integer> cs = Maybe.just(1).toCompletionStage();

// Signals NoSuchElementException
CompletionStage<Integer> cs = Maybe.empty().toCompletionStage();

// Signals 10
CompletionStage<Integer> cs = Maybe.empty().toCompletionStage(10);

// Signals 10
CompletionStage<String> cs2 = Completable.empty().toCompletionStage(10);
```

:information_source: **Further references:** Issue #6776, PR #6783

blockingStream

Available in: ● `Flowable`, ● `Observable`, ○ `Maybe`, ○ `Single`, ○ `Completable`

Exposes a `Flowable` or an `Observable` as a blocking `java.util.stream.Stream`.

:bulb: Streams are expected to have zero to N (possibly infinite) elements and thus there is no good reason to stream a `Maybe`, `Single` or `Completable`. Use the appropriate `blockingGet` and `blockingAwait` methods instead.

:warning: Abandoning a `Stream` may cause leaks or computations running indefinitely. It is recommended one closes the `Stream` manually or via the **try-with-resources** construct of Java 7+.

- `Flowable.blockingStream()`
- `Flowable.blockingStream(int)`
- `Observable.blockingStream()`
- `Observable.blockingStream(int)`

blockingStream example

```
try (Stream stream = Flowable.range(1, 10)
    .subscribeOn(Schedulers.computation())
    .blockingStream()) {

   stream.limit(5)
```

```
    .forEach(System.out::println);
}
```

:information_source: **Further references:** Issue #6776, PR #6779, PR #6797

flatMapStream, concatMapStream

Available in: ● `Flowable`, ● `Observable`, ○ `Maybe`, ○ `Single`, ○ `Completable`, ● `ParallelFlowable`

Maps each upstream item to a `java.util.stream.Stream` and emits those inner items, in sequence, non-overlappingly to the downstream.

:bulb: `flatMapStream` and `concatMapStream` are essentially the same operators because consuming a `Stream` is a sequential (and perhaps blocking) operation, thus there is no way two or more distinct `Stream` s could get interleaved.

:bulb: Since a `Stream` can be exposed as both backpressure-supporting `Flowable` or a backpressure-unsupporting `Observable`, the equivalent operators for `Maybe` and `Single` are called `flattenStreamAsFlowable` and `flattenStreamAsObservable`.

:warning: RxJava 3 doesn't accept the primitive `Stream` types (such as `IntStream` and `DoubleStream`). These streams have to be converted into their boxed variants via their `boxed()` method. Since RxJava uses reference types, there is no way to optimize the interoperation with the primitive streams.

- `Flowable.concatMapStream(Function)`
- `Flowable.concatMapStream(Function, int)`
- `Flowable.flatMapStream(Function)`
- `Flowable.flatMapStream(Function, int)`
- `Observable.concatMapStream(Function)`
- `Observable.flatMapStream(Function)`
- `ParallelFlowable.flatMapStream(Function)`
- `ParallelFlowable.flatMapStream(Function, int)`

flatMapStream example

```
Flowable.range(1, 10)
    .flatMapStream(v -> Arrays.asList(v, v + 1).stream());

Observable.range(1, 10)
    .concatMapStream(v -> Arrays.asList(v, v + 1).stream());

Flowable.range(1, 10)
     .parallel()
     .runOn(Schedulers.computation())
     .flatMapStream(v -> Arrays.asList(v, v + 1).stream());
```

:information_source: **Further references:** Issue #6776, PR #6779, PR #6797, PR #6798

flattenStreamAsFlowable, flattenStreamAsObservable

Available in: ○ `Flowable`, ○ `Observable`, ● `Maybe`, ● `Single`, ○ `Completable`

Maps success item into a `java.util.stream.Stream` and emits those inner items.

:bulb: The equivalent operator is called `flatMapStream` in `Flowable`, `Observable` and `ParallelFlowable`.

:bulb: There are no `flattenStreamAs` methods in `Completable` because it is always empty and has no item to map.

:warning: RxJava 3 doesn't accept the primitive `Stream` types (such as `IntStream` and `DoubleStream`). These streams have to be converted into their boxed variants via their `boxed()` method. Since RxJava uses reference types, there is no way to optimize the interoperation with the primitive streams.

- `Maybe.flattenStreamAsFlowable`

- `Maybe.flattenStreamAsObservable`
- `Single.flattenStreamAsFlowable`
- `Single.flattenStreamAsObservable`

flattenStreamAs example

```
Flowable<Integer> f = Maybe.just(1)
.flattenStreamAsFlowable(v -> Arrays.asList(v, v + 1).stream());

Observable<Integer> o = Single.just(2)
.flattenStreamAsObservable(v -> IntStream.range(v, v + 10).boxed());
```

:information_source: **Further references:** Issue #6776, PR #6805

API renames

startWith

The method was ambiguous and/or inviting wrong usage in other languages. They have now been renamed to `startWithArray`, `startWithIterable` and `startWithItem`:

Flowable

- `startWithArray`
- `startWithItem`
- `startWithIterable`

Observable

- `startWithArray`
- `startWithItem`
- `startWithIterable`

:information_source: **Further references:** Issue #6122, PR #6530

onErrorResumeNext with source

The method was ambiguous and/or inviting wrong usage in other languages. They have now been renamed to `onErrorResumeWith` across all types.

- `Flowable.onErrorResumeWith()`
- `Observable.onErrorResumeWith()`
- `Maybe.onErrorResumeWith()`
- `Single.onErrorResumeWith()`
- `Completable.onErrorResumeWith()`

:information_source: **Further references:** Issue #6551, PR #6556

zipIterable

Renamed to be plain `zip` to match the naming convention with other operators (i.e., Iterable/Source versions are named plainly, array-versions receive an `Array` postfix).

- Flowable.zip()
- Observable.zip()

:information_source: **Further references:** Issue #6610, PR #6638

combineLatest with array argument

Renamed to be plain `combineLatestArray` and `combineLatestArrayDelayError` to match the naming convention with other operators (i.e., Iterable/Source versions are named plainly, array-versions receive an `Array` postfix).

- `Flowable.combineLatestArray()`
- `Flowable.combineLatestArrayDelayError()`
- `Observable.combineLatestArray()`
- `Observable.combineLatestArrayDelayError()`

:information_source: **Further references:** Issue #6820, PR #6640, PR #6838

Single.equals

Renamed to `sequenceEqual` to match the naming in the other reactive classes.

- `sequenceEqual(SingleSource, SingleSource)`

:information_source: **Further references:** Issue #6854, PR #6856

Maybe.flatMapSingleElement

The operator was confusing and has been renamed to `flatMapSingle`, replacing the original `Maybe.flatMapSingle`. The original behavior (i.e., signaling `NoSuchElementException` if the `Maybe` is empty) can be emulated via `toSingle()`.

```
source.flatMapSingle(item -> singleSource).toSingle()
```

:information_source: **Further references:** Issue #6878, PR #6891

API signature changes

Callable to Supplier

Operators accepting a `java.util.concurrent.Callable` have been changed to accept `io.reactivex.rxjava3.functions.Supplier` instead to enable the callbacks to throw any kind of exceptions.

If the operator was used with a lambda, only a recompilation is needed:

```
Flowable.defer(() -> Flowable.just(Math.random()));
```

However, if explicit implementation was used:

```
Flowable.defer(new Callable<Double>() {
    @Override
    public Double call() throws Exception {
        return Math.random();
    }
});
```

the **interface type** (`Callable` -> `Supplier`) and the **method name** (`call` -> `get`) has to be adjusted:

```
Flowable.defer(new Supplier<Double>() {
    @Override
    public Double get() throws Exception {
        return Math.random();
    }
});
```

Affected operators

(Across all reactive types, multiple overloads.)

defer	error	using
generate	buffer	collect
distinct	reduceWith	scanWith
toMap	toMultimap	

:information_source: **Further references:** PR #6511

Maybe.defaultIfEmpty

Corrected the return type to `Single` as now it is guaranteed to have a success item or an error.

- `defaultIfEmpty(T)`

:information_source: **Further references:** PR #6511

concatMapDelayError parameter order

Change the order of the `tillTheEnd` argument in `concatMapDelayError` and `concatMapEagerDelayError` to be consistent with other operators taking a boolean parameter before `prefetch` / `maxConcurrency` .

- `Flowable.concatMapDelayError()`
- `Flowable.concatMapEagerDelayError()`
- `Observable.concatMapDelayError()`
- `Observable.concatMapEagerDelayError()`

:information_source: **Further references:** Issue #6610, PR #6638

Flowable.buffer with boundary source

The signature was wrongly declared with a `Flowable` instead of a more general `Publisher` .

- `buffer(Publisher, Function)`
- `buffer(Publisher, Function, Supplier)`

:information_source: **Further references:** PR #6858

Maybe.flatMapSingle

The operator was not in line with how `flatMap` s are expected to operate (i.e., it signaled `NoSuchElementException` if the `Maybe` was empty). The `flatMapSingleElement` operator has been renamed to be the `flatMapSingle` operator.

- `flatMapSingle(Function)`

The original error behavior can be emulated via `toSingle()` :

```
source.flatMapSingle(item -> singleSource).toSingle()
```

:information_source: **Further references:** Issue #6878, PR #6891

API removals

getValues (hot sources)

The `getValue()` and `getValues(T[])` methods were a remnant from a time where `Subject` and `FlowableProcessor` was unifying all state peeking methods for every kind of subject/processor. These have been marked as `@Deprecated` in 2.x and are now removed from 3.x. They can be trivially replaced with `getValue()` if necessary, for example:

```
Object value = subject.getValue();
if (value == null) {
   return new Object[1];
}
return new Object[] { value };
```

:information_source: **Further references:** Issue #5622, PR #6516

Maybe.toSingle(T)

Use `Maybe.defaultIfEmpty(T)` instead.

:information_source: **Further references:** PR #6517

subscribe(4 arguments)

Removed from `Flowable` and `Observable` . The 4th argument, the `Subscription` / `Disposable` callback, was more or less useless. Use `Flowable.doOnSubscribe()` and `Observable.doOnSubscribe()` instead. instead.

:information_source: **Further references:** PR #6517

Single.toCompletable()

Using a better terminology instead: `ignoreElement()` .

:information_source: **Further references:** PR #6517

Completable.blockingGet()

The behavior and signature were confusing (i.e., returning `null` or a `Throwable`). Use `blockingAwait()` instead.

:information_source: **Further references:** PR #6517

test support methods

Based on user feedback, the following methods have been removed from `TestSubscriber` and `TestObserver` respectively due to being less useful outside testing RxJava itself:

assertErrorMessage	assertFailure(Predicate, T...)	assertFailureAndMessage
assertNever(Predicate)	assertNever(T)	assertNoTimeout
assertNotSubscribed	assertNotTerminated	assertSubscribed
assertTerminated	assertTimeout	assertValueSequenceOnly
assertValueSet	assertValueSetOnly	awaitCount(int, Runnable)
awaitCount(int, Runnable, long)	awaitTerminalEvent	awaitTerminalEvent(long TimeUnit)
clearTimeout	completions	errorCount
errors	getEvents	isTerminated
isTimeout	lastThread	valueCount
assertOf		

:information_source: **Further references:** Issue #6153, PR #6526

replay with Scheduler

The `replay(Scheduler)` and other overloads were carried over from the original Rx.NET API set but appears to be unused. Most use cases capture the connectable anyway so there is no much benefit from inlining an `observeOn` into a connectable:

```
ConnectableFlowable<Integer> connectable = source.replay();

Flowable<Integr> flowable = connectable.observeOn(Schedulers.io());

// hand flowable to consumers
flowable.subscribe();

connectable.connect();
```

:information_source: **Further references:** PR #6539

dematerialize()

The `Flowable.dematerialize()` and `Observable.dematerialize()` were inherently type-unsafe and have been removed. In Rx.NET, the extension methods allowed `dematerialize()` to be applied to `Observable<Notification<T>>` only, but there is no way for doing it in Java as it has no extension methods and one can't restrict a method to appear only with a certain type argument scheme.

Use `deserialize(Function)` instead.

```
Observable<Notification<Integer>> source = ...

Observable<Integer> result = source.dematerialize(v -> v);
```

:information_source: **Further references:** PR #6539

onExceptionResumeNext

The operator was apparently not used anywhere and has been removed from all types. It's function can be emulated via `onErrorResumeNext`:

```
source.onErrorResumeNext(
    error -> error instanceof Exception
        ? fallback : Obserable.error(error))
```

:information_source: **Further references:** Issue #6554, PR #6564, PR #6844

buffer with boundary supplier

This operator did not see much use and have been removed from `Flowable` and `Observable`. It can be emulated with the plain sourced version:

```
source.buffer(Observable.defer(supplier).take(1).repeat())
```

:information_source: **Further references:** Issue #6555, PR #6564

combineLatest with varags

Both the vararg overloaded versions of `combineLatest` and `combineLatestDelayError` were awkward to use from other JVM languages and have been removed. Use `combineLatestArray` and `combineLatestArrayDelayError` instead.

:information_source: **Further references:** Issue #6634, PR #6635

zip with nested source

Zip requires a known number of sources to work with. These overloads were just collecting up the inner sources for another overload. Removed from both `Flowable` and `Observable`. They can be emulated via composition:

```
Observable<Observable<Integer>> sources = ...

sources.toList().flatMapObservable(list -> Observable.zip(list, zipper));
```

:information_source: **Further references:** PR #6638

fromFuture with scheduler

These were just convenience overloads for `fromFuture().subscribeOn()` all across. Apply `subscribeOn` explicitly from now on.

```
Flowable.fromFuture(future).subscribeOn(Schedulers.io());

Flowable.fromFuture(future, 5, TimeUnit.SECONDS)
    .subscribeOn(Schedulers.io());
```

:information_source: **Further references:** Issue #6811, PR #6814

Observable.concatMapIterable with buffer parameter

This overload had no effect because there is no buffering happening inside the operator (unlike in the `Flowable` variant). Use the `Observable.concatMapIterable(Function)` overload instead.

:information_source: **Further references:** Issue #6828, PR #6837

Interoperation

Reactive Streams

RxJava 3 still follows the Reactive Streams specification and as such, the `io.reactivex.rxjava3.core.Flowable` is a compatible source for any 3rd party solution accepting an `org.reactivestreams.Publisher` as input.

`Flowable` can also wrap any such `org.reactivestreams.Publisher` in return.

:information_source: Note that it is possible to interface RxJava's 2.x `Flowable` and 3.x `Flowable` this way, however, due to the specification requirements, this involves extra overhead. Instead, one should use a dedicated interoperation library.

RxJava 1.x

RxJava is more than 7 years old at this moment and many users are still stuck with 3rd party tools or libraries only supporting the RxJava 1 line.

To help with the situation, and also help with a gradual migration from 1.x to 3.x, an external interop library is provided:

https://github.com/akarnokd/RxJavaInterop#rxjavainterop

RxJava 2.x

Migration from 2.x to 3.x could also be cumbersome as well as difficult because the 2.x line also amassed an ecosystem of tools and libraries of its own, which may take time to provide a native 3.x versions.

RxJava 3.x was structured, both in code and in Maven coordinates, to allow the existence of both 2.x and 3.x code side by side (or even have all 3 major versions at once).

There is limited interoperation between the `Flowable`s through the Reactive Streams `Publisher` interface (although not recommended due to extra overheads), however, there is no direct way for a 2.x `Observable` to talk to a 3.x `Observable` as they are completely separate types.

To help with the situation, and also help with a gradual migration from 2.x to 3.x, an external interop library is provided:

https://github.com/akarnokd/RxJavaBridge#rxjavabridge

Java 9

The move to a Java 8 baseline was enabled by Android's improved (and upcoming) desugaring toolset.

Unfortunately, there was no indication if and when this tooling would support Java 9, more specifically, the `java.util.concurrent.Flow` interfaces imported and standardized from the 3rd party Reactive Streams specification.

There is a semi-hidden `org.reactivestreams.FlowAdapter` class in the Reactive Streams library that could be used for conversion in-between, but yet again, it adds some extra overhead.

Therefore, an external, native interoperation library is provided to convert between `java.util.concurrent.Flow.Publisher` and `io.reactivex.rxjava3.core.Flowable` as well as `java.util.concurrent.Flow.Processor` and `io.reactivex.rxjava3.processors.FlowableProcessor`.

https://github.com/akarnokd/RxJavaJdk9Interop#rxjavajdk9interop

:bulb: Conversion to other RxJava 3 reactive types are not supported and the user is expected to apply the appropriate RxJava 3 conversion method.

Swing

Since the Graphical User Interface library Swing is not part of the Android platform, the desktop users of the JDK have to resort to an external library to work with GUI components and their event sources:

https://github.com/akarnokd/RxJavaSwing#rxjavaswing

Project Loom

There is a project in the works at Oracle trying to solve the asynchronous programming problems in a different way than RxJava and **reactive programming** has been doing it for more than a decade.

The idea is to have the user code appear to be imperative and blocking, but the JVM will make it so that actual, and resource-limited, native OS threads don't get blocked.

:warning: Note that *Project Loom* is currently incomplete and keeps changing its surface API from preview to preview. Every new preview version may require rework in the respective user and interop implementations again and again.

It could be years till the design and implementation of *Project Loom* becomes mainstream in the JDK, and perhaps even more time until Android picks it up. Therefore, to allow early experimentation, an external library is provided to allow working with generators written in an imperative and (virtually) blocking fashion:

https://github.com/akarnokd/RxJavaFiberInterop#rxjavafiberinterop

:bulb: Since Project Loom offers a transparent way of turning blocking operations (such as `CountDownLatch.await()`) into economic virtual thread suspensions, there is no need to provide any specific conversion method from RxJava 3 to *Project Loom*; executing any of the standard RxJava `blockingXXX` method in a virtual thread automatically benefits from this transparent suspension.

Miscellaneous

Changelog of the individual release candidates

- 3.0.0-RC0
- 3.0.0-RC1
- 3.0.0-RC2
- 3.0.0-RC3
- 3.0.0-RC4
- 3.0.0-RC5
- 3.0.0-RC6
- 3.0.0-RC7
- 3.0.0-RC8
- 3.0.0-RC9