RxJava 2.0 has been completely rewritten from scratch on top of the Reactive-Streams specification. The specification itself has evolved out of RxJava 1.x and provides a common baseline for reactive systems and libraries.

Because Reactive-Streams has a different architecture, it mandates changes to some well known RxJava types. This wiki page attempts to summarize what has changed and describes how to rewrite 1.x code into 2.x code.

For technical details on how to write operators for 2.x, please visit the Writing Operators wiki page.

Contents

Maven address and base package

To allow having RxJava 1.x and RxJava 2.x side-by-side, RxJava 2.x is under the maven coordinates `io.reactivex.rxjava2:rxjava:2.x.y` and classes are accessible below `io.reactivex`.

Users switching from 1.x to 2.x have to re-organize their imports, but carefully.

Javadoc

The official Javadoc pages for 2.x is hosted at http://reactivex.io/RxJava/2.x/javadoc/

Nulls

RxJava 2.x no longer accepts `null` values and the following will yield `NullPointerException` immediately or as a signal to downstream:

```java
Observable.just(null);

Single.just(null);

Observable.fromCallable(() -> null)
    .subscribe(System.out::println, Throwable::printStackTrace);

Observable.just(1).map(v -> null)
    .subscribe(System.out::println, Throwable::printStackTrace);
```

This means that `Observable<Void>` can no longer emit any values but only terminate normally or with an exception. API designers may instead choose to define `Observable<Object>` with no guarantee on what `Object` will be (which should be irrelevant anyway). For example, if one needs a signaller-like source, a shared enum can be defined and its solo instance `onNext` 'd:

```java
enum Irrelevant { INSTANCE; }

Observable<Object> source = Observable.create((ObservableEmitter<Object> emitter) ->
{
   System.out.println("Side-effect 1");
   emitter.onNext(Irrelevant.INSTANCE);

   System.out.println("Side-effect 2");
   emitter.onNext(Irrelevant.INSTANCE);

   System.out.println("Side-effect 3");
   emitter.onNext(Irrelevant.INSTANCE);
});

source.subscribe(e -> { /* Ignored. */ }, Throwable::printStackTrace);
```

Observable and Flowable

A small regret about introducing backpressure in RxJava 0.x is that instead of having a separate base reactive class, the `Observable` itself was retrofitted. The main issue with backpressure is that many hot sources, such as UI events, can't be reasonably backpressured and cause unexpected `MissingBackpressureException` (i.e., beginners don't expect them).

We try to remedy this situation in 2.x by having `io.reactivex.Observable` non-backpressured and the new `io.reactivex.Flowable` be the backpressure-enabled base reactive class.

The good news is that operator names remain (mostly) the same. The bad news is that one should be careful when performing 'organize imports' as it may select the non-backpressured `io.reactivex.Observable` unintended.

Which type to use?

When architecting dataflows (as an end-consumer of RxJava) or deciding upon what type your 2.x compatible library should take and return, you can consider a few factors that should help you avoid problems down the line such as `MissingBackpressureException` or `OutOfMemoryError`.

When to use Observable

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

When to use Flowable

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

Single

The 2.x `Single` reactive base type, which can emit a single `onSuccess` or `onError` has been redesigned from scratch. Its architecture now derives from the Reactive-Streams design. Its consumer type (`rx.Single.SingleSubscriber<T>`) has been changed from being a class that accepts `rx.Subscription` resources to be an interface `io.reactivex.SingleObserver<T>` that has only 3 methods:

```
interface SingleObserver<T> {
    void onSubscribe(Disposable d);
    void onSuccess(T value);
    void onError(Throwable error);
}
```

and follows the protocol `onSubscribe (onSuccess | onError)?`.

Completable

The `Completable` type remains largely the same. It was already designed along the Reactive-Streams style for 1.x so no user-level changes there.

Similar to the naming changes, `rx.Completable.CompletableSubscriber` has become `io.reactivex.CompletableObserver` with `onSubscribe(Disposable)`:

```
interface CompletableObserver<T> {
    void onSubscribe(Disposable d);
    void onComplete();
```

```
    void onError(Throwable error);
}
```

and still follows the protocol `onSubscribe (onComplete | onError)?`.

Maybe

RxJava 2.0.0-RC2 introduced a new base reactive type called `Maybe`. Conceptually, it is a union of `Single` and `Completable` providing the means to capture an emission pattern where there could be 0 or 1 item or an error signalled by some reactive source.

The `Maybe` class is accompanied by `MaybeSource` as its base interface type, `MaybeObserver<T>` as its signal-receiving interface and follows the protocol `onSubscribe (onSuccess | onError | onComplete)?`. Because there could be at most 1 element emitted, the `Maybe` type has no notion of backpressure (because there is no buffer bloat possible as with unknown length `Flowable`s or `Observable`s.

This means that an invocation of `onSubscribe(Disposable)` is potentially followed by one of the other `onXXX` methods. Unlike `Flowable`, if there is only a single value to be signalled, only `onSuccess` is called and `onComplete` is not.

Working with this new base reactive type is practically the same as the others as it offers a modest subset of the `Flowable` operators that make sense with a 0 or 1 item sequence.

```
Maybe.just(1)
.map(v -> v + 1)
.filter(v -> v == 1)
.defaultIfEmpty(2)
.test()
.assertResult(2);
```

Base reactive interfaces

Following the style of extending the Reactive-Streams `Publisher<T>` in `Flowable`, the other base reactive classes now extend similar base interfaces (in package `io.reactivex`):

```
interface ObservableSource<T> {
    void subscribe(Observer<? super T> observer);
}

interface SingleSource<T> {
    void subscribe(SingleObserver<? super T> observer);
}

interface CompletableSource {
    void subscribe(CompletableObserver observer);
}

interface MaybeSource<T> {
```

```
    void subscribe(MaybeObserver<? super T> observer);
}
```

Therefore, many operators that required some reactive base type from the user now accept `Publisher` and `XSource`:

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

Observable<R> flatMap(Function<? super T, ? extends ObservableSource<? extends R>>
mapper);
```

By having `Publisher` as input this way, you can compose with other Reactive-Streams compliant libraries without the need to wrap them or convert them into `Flowable` first.

If an operator has to offer a reactive base type, however, the user will receive the full reactive class (as giving out an `XSource` is practically useless as it doesn't have operators on it):

```
Flowable<Flowable<Integer>> windows = source.window(5);

source.compose((Flowable<T> flowable) ->
    flowable
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread()));
```

Subjects and Processors

In the Reactive-Streams specification, the `Subject`-like behavior, namely being a consumer and supplier of events at the same time, is done by the `org.reactivestreams.Processor` interface. As with the `Observable` / `Flowable` split, the backpressure-aware, Reactive-Streams compliant implementations are based on the `FlowableProcessor<T>` class (which extends `Flowable` to give a rich set of instance operators). An important change regarding `Subject`s (and by extension, `FlowableProcessor`) that they no longer support `T -> R` like conversion (that is, input is of type `T` and the output is of type `R`). (We never had a use for it in 1.x and the original `Subject<T, R>` came from .NET where there is a `Subject<T>` overload because .NET allows the same class name with a different number of type arguments.)

The `io.reactivex.subjects.AsyncSubject`, `io.reactivex.subjects.BehaviorSubject`, `io.reactivex.subjects.PublishSubject`, `io.reactivex.subjects.ReplaySubject` and `io.reactivex.subjects.UnicastSubject` in 2.x don't support backpressure (as part of the 2.x `Observable` family).

The `io.reactivex.processors.AsyncProcessor`, `io.reactivex.processors.BehaviorProcessor`, `io.reactivex.processors.PublishProcessor`, `io.reactivex.processors.ReplayProcessor` and `io.reactivex.processors.UnicastProcessor` are backpressure-aware. The `BehaviorProcessor` and `PublishProcessor` don't coordinate requests (use `Flowable.publish()` for that) of their downstream subscribers and will signal them `MissingBackpressureException` if the downstream can't keep up. The other `XProcessor` types honor backpressure of their downstream subscribers but otherwise, when subscribed to a source (optional), they consume it in an unbounded manner (requesting `Long.MAX_VALUE`).

TestSubject

The 1.x `TestSubject` has been dropped. Its functionality can be achieved via `TestScheduler`, `PublishProcessor` / `PublishSubject` and `observeOn(testScheduler)` /scheduler parameter.

```
TestScheduler scheduler = new TestScheduler();
PublishSubject<Integer> ps = PublishSubject.create();

TestObserver<Integer> ts = ps.delay(1000, TimeUnit.MILLISECONDS, scheduler)
.test();

ts.assertEmpty();

ps.onNext(1);

scheduler.advanceTimeBy(999, TimeUnit.MILLISECONDS);

ts.assertEmpty();

scheduler.advanceTimeBy(1, TimeUnit.MILLISECONDS);

ts.assertValue(1);
```

SerializedSubject

The `SerializedSubject` is no longer a public class. You have to use `Subject.toSerialized()` and `FlowableProcessor.toSerialized()` instead.

Other classes

The `rx.observables.ConnectableObservable` is now `io.reactivex.observables.ConnectableObservable<T>` and `io.reactivex.flowables.ConnectableFlowable<T>`.

GroupedObservable

The `rx.observables.GroupedObservable` is now `io.reactivex.observables.GroupedObservable<T>` and `io.reactivex.flowables.GroupedFlowable<T>`.

In 1.x, you could create an instance with `GroupedObservable.from()` which was used internally by 1.x. In 2.x, all use cases now extend `GroupedObservable` directly thus the factory methods are no longer available; the whole class is now abstract.

You can extend the class and add your own custom `subscribeActual` behavior to achieve something similar to the 1.x features:

```java
class MyGroup<K, V> extends GroupedObservable<K, V> {
    final K key;

    final Subject<V> subject;

    public MyGroup(K key) {
        this.key = key;
        this.subject = PublishSubject.create();
    }

    @Override
    public T getKey() {
        return key;
    }

    @Override
    protected void subscribeActual(Observer<? super T> observer) {
        subject.subscribe(observer);
    }
}
```

(The same approach works with `GroupedFlowable` as well.)

Functional interfaces

Because both 1.x and 2.x is aimed at Java 6+, we can't use the Java 8 functional interfaces such as `java.util.function.Function`. Instead, we defined our own functional interfaces in 1.x and 2.x follows this tradition.

One notable difference is that all our functional interfaces now define `throws Exception`. This is a large convenience for consumers and mappers that otherwise throw and would need `try-catch` to transform or suppress a checked exception.

```java
Flowable.just("file.txt")
.map(name -> Files.readLines(name))
.subscribe(lines -> System.out.println(lines.size()), Throwable::printStackTrace);
```

If the file doesn't exist or can't be read properly, the end consumer will print out `IOException` directly. Note also the `Files.readLines(name)` invoked without try-catch.

Actions

As the opportunity to reduce component count, 2.x doesn't define `Action3` - `Action9` and `ActionN` (these were unused within RxJava itself anyway).

The remaining action interfaces were named according to the Java 8 functional types. The no argument `Action0` is replaced by the `io.reactivex.functions.Action` for the operators and `java.lang.Runnable` for the `Scheduler` methods. `Action1` has been renamed to `Consumer` and `Action2` is called `BiConsumer`. `ActionN` is replaced by the `Consumer<Object[]>` type declaration.

Functions

We followed the naming convention of Java 8 by defining `io.reactivex.functions.Function` and `io.reactivex.functions.BiFunction`, plus renaming `Func3` - `Func9` into `Function3` - `Function9` respectively. The `FuncN` is replaced by the `Function<Object[], R>` type declaration.

In addition, operators requiring a predicate no longer use `Func1<T, Boolean>` but have a separate, primitive-returning type of `Predicate<T>` (allows better inlining due to no autoboxing).

Subscriber

The Reactive-Streams specification has its own Subscriber as an interface. This interface is lightweight and combines request management with cancellation into a single interface `org.reactivestreams.Subscription` instead of having `rx.Producer` and `rx.Subscription` separately. This allows creating stream consumers with less internal state than the quite heavy `rx.Subscriber` of 1.x.

```
Flowable.range(1, 10).subscribe(new Subscriber<Integer>() {
    @Override
    public void onSubscribe(Subscription s) {
        s.request(Long.MAX_VALUE);
    }

    @Override
    public void onNext(Integer t) {
        System.out.println(t);
    }

    @Override
    public void onError(Throwable t) {
        t.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("Done");
    }
});
```

Due to the name conflict, replacing the package from `rx` to `org.reactivestreams` is not enough. In addition, `org.reactivestreams.Subscriber` has no notion of adding resources to it, cancelling it or requesting from the outside.

To bridge the gap we defined abstract classes `DefaultSubscriber`, `ResourceSubscriber` and `DisposableSubscriber` (plus their `XObserver` variants) for `Flowable` (and `Observable`) respectively that offers resource tracking support (of `Disposable` s) just like `rx.Subscriber` and can be cancelled/disposed externally via `dispose()` :

```
ResourceSubscriber<Integer> subscriber = new ResourceSubscriber<Integer>() {
    @Override
```

```java
    public void onStart() {
        request(Long.MAX_VALUE);
    }

    @Override
    public void onNext(Integer t) {
        System.out.println(t);
    }

    @Override
    public void onError(Throwable t) {
        t.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("Done");
    }
};

Flowable.range(1, 10).delay(1, TimeUnit.SECONDS).subscribe(subscriber);

subscriber.dispose();
```

Note also that due to Reactive-Streams compatibility, the method `onCompleted` has been renamed to `onComplete` without the trailing `d`.

Since 1.x `Observable.subscribe(Subscriber)` returned `Subscription`, users often added the `Subscription` to a `CompositeSubscription` for example:

```java
CompositeSubscription composite = new CompositeSubscription();

composite.add(Observable.range(1, 5).subscribe(new TestSubscriber<Integer>()));
```

Due to the Reactive-Streams specification, `Publisher.subscribe` returns void and the pattern by itself no longer works in 2.0. To remedy this, the method `E subscribeWith(E subscriber)` has been added to each base reactive class which returns its input subscriber/observer as is. With the two examples before, the 2.x code can now look like this since `ResourceSubscriber` implements `Disposable` directly:

```java
CompositeDisposable composite2 = new CompositeDisposable();

composite2.add(Flowable.range(1, 5).subscribeWith(subscriber));
```

Calling request from onSubscribe/onStart

Note that due to how request management works, calling `request(n)` from `Subscriber.onSubscribe` or `ResourceSubscriber.onStart` may trigger calls to `onNext` immediately before the `request()` call itself returns to the `onSubscribe` / `onStart` method of yours:

```java
Flowable.range(1, 3).subscribe(new Subscriber<Integer>() {

    @Override
    public void onSubscribe(Subscription s) {
        System.out.println("OnSubscribe start");
        s.request(Long.MAX_VALUE);
        System.out.println("OnSubscribe end");
    }

    @Override
    public void onNext(Integer v) {
        System.out.println(v);
    }

    @Override
    public void onError(Throwable e) {
        e.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("Done");
    }
});
```

This will print:

```
OnSubscribe start
1
2
3
Done
OnSubscribe end
```

The problem comes when one does some initialization in `onSubscribe` / `onStart` after calling `request` there and `onNext` may or may not see the effects of the initialization. To avoid this situation, make sure you call `request` **after** all initialization have been done in `onSubscribe` / `onStart` .

This behavior differs from 1.x where a `request` call went through a deferred logic that accumulated requests until an upstream `Producer` arrived at some time (This nature adds overhead to all operators and consumers in 1.x.) In 2.x, there is always a `Subscription` coming down first and 90% of the time there is no need to defer requesting.

Subscription

In RxJava 1.x, the interface `rx.Subscription` was responsible for stream and resource lifecycle management, namely unsubscribing a sequence and releasing general resources such as scheduled tasks. The Reactive-Streams specification took this name for specifying an interaction point between a source and a consumer: `org.reactivestreams.Subscription` allows requesting a positive amount from the upstream and allows cancelling the sequence.

To avoid the name clash, the 1.x `rx.Subscription` has been renamed into `io.reactivex.Disposable` (somewhat resembling .NET's own IDisposable).

Because Reactive-Streams base interface, `org.reactivestreams.Publisher` defines the `subscribe()` method as `void`, `Flowable.subscribe(Subscriber)` no longer returns any `Subscription` (or `Disposable`). The other base reactive types also follow this signature with their respective subscriber types.

The other overloads of `subscribe` now return `Disposable` in 2.x.

The original `Subscription` container types have been renamed and updated

- `CompositeSubscription` to `CompositeDisposable`
- `SerialSubscription` and `MultipleAssignmentSubscription` have been merged into `SerialDisposable`. The `set()` method disposes the old value and `replace()` method does not.
- `RefCountSubscription` has been removed.

Backpressure

The Reactive-Streams specification mandates operators supporting backpressure, specifically via the guarantee that they don't overflow their consumers when those don't request. Operators of the new `Flowable` base reactive type now consider downstream request amounts properly, however, this doesn't mean `MissingBackpressureException` is gone. The exception is still there but this time, the operator that can't signal more `onNext` will signal this exception instead (allowing better identification of who is not properly backpressured).

As an alternative, the 2.x `Observable` doesn't do backpressure at all and is available as a choice to switch over.

Reactive-Streams compliance

updated in 2.0.7

The `Flowable`-based sources and operators are, as of 2.0.7, fully Reactive-Streams version 1.0.0 specification compliant.

Before 2.0.7, the operator `strict()` had to be applied in order to achieve the same level of compliance. In 2.0.7, the operator `strict()` returns `this`, is deprecated and will be removed completely in 2.1.0.

As one of the primary goals of RxJava 2, the design focuses on performance and in order enable it, RxJava 2.0.7 adds a custom `io.reactivex.FlowableSubscriber` interface (extends `org.reactivestreams.Subscriber`) but adds no new methods to it. The new interface is **constrained to RxJava 2** and represents a consumer to `Flowable` that is able to work in a mode that relaxes the Reactive-Streams version 1.0.0 specification in rules §1.3, §2.3, §2.12 and §3.9:

- §1.3 relaxation: `onSubscribe` may run concurrently with `onNext` in case the `FlowableSubscriber` calls `request()` from inside `onSubscribe` and it is the resposibility of `FlowableSubscriber` to ensure thread-safety between the remaining instructions in `onSubscribe` and `onNext`.
- §2.3 relaxation: calling `Subscription.cancel` and `Subscription.request` from `FlowableSubscriber.onComplete()` or `FlowableSubscriber.onError()` is considered a no-operation.

- §2.12 relaxation: if the same `FlowableSubscriber` instance is subscribed to multiple sources, it must ensure its `onXXX` methods remain thread safe.
- §3.9 relaxation: issuing a non-positive `request()` will not stop the current stream but signal an error via `RxJavaPlugins.onError`.

From a user's perspective, if one was using the the `subscribe` methods other than `Flowable.subscribe(Subscriber<? super T>)`, there is no need to do anything regarding this change and there is no extra penalty for it.

If one was using `Flowable.subscribe(Subscriber<? super T>)` with the built-in RxJava `Subscriber` implementations such as `DisposableSubscriber`, `TestSubscriber` and `ResourceSubscriber`, there is a small runtime overhead (one `instanceof` check) associated when the code is not recompiled against 2.0.7.

If a custom class implementing `Subscriber` was employed before, subscribing it to a `Flowable` adds an internal wrapper that ensures observing the Flowable is 100% compliant with the specification at the cost of some per-item overhead.

In order to help lift these extra overheads, a new method `Flowable.subscribe(FlowableSubscriber<? super T>)` has been added which exposes the original behavior from before 2.0.7. It is recommended that new custom consumer implementations extend `FlowableSubscriber` instead of just `Subscriber`.

Runtime hooks

The 2.x redesigned the `RxJavaPlugins` class which now supports changing the hooks at runtime. Tests that want to override the schedulers and the lifecycle of the base reactive types can do it on a case-by-case basis through callback functions.

The class-based `RxJavaObservableHook` and friends are now gone and `RxJavaHooks` functionality is incorporated into `RxJavaPlugins`.

Error handling

One important design requirement for 2.x is that no `Throwable` errors should be swallowed. This means errors that can't be emitted because the downstream's lifecycle already reached its terminal state or the downstream cancelled a sequence which was about to emit an error.

Such errors are routed to the `RxJavaPlugins.onError` handler. This handler can be overridden with the method `RxJavaPlugins.setErrorHandler(Consumer<Throwable>)`. Without a specific handler, RxJava defaults to printing the `Throwable`'s stacktrace to the console and calls the current thread's uncaught exception handler.

On desktop Java, this latter handler does nothing on an `ExecutorService` backed `Scheduler` and the application can keep running. However, Android is more strict and terminates the application in such uncaught exception cases.

If this behavior is desirable can be debated, but in any case, if you want to avoid such calls to the uncaught exception handler, the **final application** that uses RxJava 2 (directly or transitively) should set a no-op handler:

```
// If Java 8 lambdas are supported
RxJavaPlugins.setErrorHandler(e -> { });
```

```
// If no Retrolambda or Jack
RxJavaPlugins.setErrorHandler(Functions.<Throwable>emptyConsumer());
```

It is not advised intermediate libraries change the error handler outside their own testing environment.

Unfortunately, RxJava can't tell which of these out-of-lifecycle, undeliverable exceptions should or shouldn't crash your app. Identifying the source and reason for these exceptions can be tiresome, especially if they originate from a source and get routed to `RxJavaPlugins.onError` somewhere lower the chain.

Therefore, 2.0.6 introduces specific exception wrappers to help distinguish and track down what was happening the time of the error:

- `OnErrorNotImplementedException` : reintroduced to detect when the user forgot to add error handling to `subscribe()` .
- `ProtocolViolationException` : indicates a bug in an operator
- `UndeliverableException` : wraps the original exception that can't be delivered due to lifecycle restrictions on a `Subscriber` / `Observer` . It is automatically applied by `RxJavaPlugins.onError` with intact stacktrace that may help find which exact operator rerouted the original error.

If an undeliverable exception is an instance/descendant of `NullPointerException` , `IllegalStateException` (`UndeliverableException` and `ProtocolViolationException` extend this), `IllegalArgumentException` , `CompositeException` , `MissingBackpressureException` or `OnErrorNotImplementedException` , the `UndeliverableException` wrapping doesn't happen.

In addition, some 3rd party libraries/code throw when they get interrupted by a cancel/dispose call which leads to an undeliverable exception most of the time. Internal changes in 2.0.6 now consistently cancel or dispose a `Subscription` / `Disposable` before cancelling/disposing a task or worker (which causes the interrupt on the target thread).

```
// in some library
try {
   doSomethingBlockingly()
} catch (InterruptedException ex) {
   // check if the interrupt is due to cancellation
   // if so, no need to signal the InterruptedException
   if (!disposable.isDisposed()) {
      observer.onError(ex);
   }
}
```

If the library/code already did this, the undeliverable `InterruptedException` s should stop now. If this pattern was not employed before, we encourage updating the code/library in question.

If one decides to add a non-empty global error consumer, here is an example that manages the typical undeliverable exceptions based on whether they represent a likely bug or an ignorable application/network state:

```
RxJavaPlugins.setErrorHandler(e -> {
    if (e instanceof UndeliverableException) {
        e = e.getCause();
    }
    if ((e instanceof IOException) || (e instanceof SocketException)) {
```

```
            // fine, irrelevant network problem or API that throws on cancellation
            return;
        }
        if (e instanceof InterruptedException) {
            // fine, some blocking code was interrupted by a dispose call
            return;
        }
        if ((e instanceof NullPointerException) || (e instanceof
IllegalArgumentException)) {
            // that's likely a bug in the application
            Thread.currentThread().getUncaughtExceptionHandler()
                .handleException(Thread.currentThread(), e);
            return;
        }
        if (e instanceof IllegalStateException) {
            // that's a bug in RxJava or in a custom operator
            Thread.currentThread().getUncaughtExceptionHandler()
                .handleException(Thread.currentThread(), e);
            return;
        }
        Log.warning("Undeliverable exception received, not sure what to do", e);
});
```

Schedulers

The 2.x API still supports the main default scheduler types: `computation`, `io`, `newThread` and `trampoline`, accessible through `io.reactivex.schedulers.Schedulers` utility class.

The `immediate` scheduler is not present in 2.x. It was frequently misused and didn't implement the `Scheduler` specification correctly anyway; it contained blocking sleep for delayed action and didn't support recursive scheduling at all. Use `Schedulers.trampoline()` instead.

The `Schedulers.test()` has been removed as well to avoid the conceptional difference with the rest of the default schedulers. Those return a "global" scheduler instance whereas `test()` returned always a new instance of the `TestScheduler`. Test developers are now encouraged to simply `new TestScheduler()` in their code.

The `io.reactivex.Scheduler` abstract base class now supports scheduling tasks directly without the need to create and then destroy a `Worker` (which is often forgotten):

```
public abstract class Scheduler {

    public Disposable scheduleDirect(Runnable task) { ... }

    public Disposable scheduleDirect(Runnable task, long delay, TimeUnit unit) { ...
}

    public Disposable scheduleDirectPeriodically(Runnable task, long initialDelay,
        long period, TimeUnit unit) { ... }

    public long now(TimeUnit unit) { ... }
```

```
    // ... rest is the same: lifecycle methods, worker creation
}
```

The main purpose is to avoid the tracking overhead of the `Worker` s for typically one-shot tasks. The methods have a default implementation that reuses `createWorker` properly but can be overridden with more efficient implementations if necessary.

The method that returns the scheduler's own notion of current time, `now()` has been changed to accept a `TimeUnit` to indicate the unit of measure.

Entering the reactive world

One of the design flaws of RxJava 1.x was the exposure of the `rx.Observable.create()` method that while powerful, not the typical operator you want to use to enter the reactive world. Unfortunately, so many depend on it that we couldn't remove or rename it.

Since 2.x is a fresh start, we won't make that mistake again. Each reactive base type `Flowable` , `Observable` , `Single` , `Maybe` and `Completable` feature a safe `create` operator that does the right thing regarding backpressure (for `Flowable`) and cancellation (all):

```
Flowable.create((FlowableEmitter<Integer> emitter) -> {
    emitter.onNext(1);
    emitter.onNext(2);
    emitter.onComplete();
}, BackpressureStrategy.BUFFER);
```

Practically, the 1.x `fromEmitter` (formerly `fromAsync`) has been renamed to `Flowable.create` . The other base reactive types have similar `create` methods (minus the backpressure strategy).

Leaving the reactive world

Apart from subscribing to the base types with their respective consumers (`Subscriber` , `Observer` , `SingleObserver` , `MaybeObserver` and `CompletableObserver`) and functional-interface based consumers (such as `subscribe(Consumer<T>, Consumer<Throwable>, Action)`), the formerly separate 1.x `BlockingObservable` (and similar classes for the others) has been integrated with the main reactive type. Now you can directly block for some results by invoking a `blockingX` operation directly:

```
List<Integer> list = Flowable.range(1, 100).toList().blockingGet(); // toList()
returns Single

Integer i = Flowable.range(100, 100).blockingLast();
```

(The reason for this is twofold: performance and ease of use of the library as a synchronous Java 8 Streams-like processor.)

Another significant difference between `rx.Subscriber` (and co) and `org.reactivestreams.Subscriber` (and co) is that in 2.x, your `Subscriber` s and `Observer` s are not allowed to throw anything but fatal exceptions (see `Exceptions.throwIfFatal()`). (The Reactive-Streams specification allows throwing `NullPointerException` if the `onSubscribe` , `onNext` or `onError` receives a `null` value, but RxJava doesn't let `null` s in any way.) This means the following code is no longer legal:

```
Subscriber<Integer> subscriber = new Subscriber<Integer>() {
    @Override
    public void onSubscribe(Subscription s) {
        s.request(Long.MAX_VALUE);
    }

    public void onNext(Integer t) {
        if (t == 1) {
            throw new IllegalArgumentException();
        }
    }

    public void onError(Throwable e) {
        if (e instanceof IllegalArgumentException) {
            throw new UnsupportedOperationException();
        }
    }

    public void onComplete() {
        throw new NoSuchElementException();
    }
};

Flowable.just(1).subscribe(subscriber);
```

The same applies to `Observer` , `SingleObserver` , `MaybeObserver` and `CompletableObserver` .

Since many of the existing code targeting 1.x do such things, the method `safeSubscribe` has been introduced that does handle these non-conforming consumers.

Alternatively, you can use the `subscribe(Consumer<T>, Consumer<Throwable>, Action)` (and similar) methods to provide a callback/lambda that can throw:

```
Flowable.just(1)
.subscribe(
    subscriber::onNext,
    subscriber::onError,
    subscriber::onComplete,
    subscriber::onSubscribe
);
```

Testing

Testing RxJava 2.x works the same way as it does in 1.x. `Flowable` can be tested with `io.reactivex.subscribers.TestSubscriber` whereas the non-backpressured `Observable` , `Single` , `Maybe` and `Completable` can be tested with `io.reactivex.observers.TestObserver`.

test() "operator"

To support our internal testing, all base reactive types now feature `test()` methods (which is a huge convenience for us) returning `TestSubscriber` or `TestObserver`:

```
TestSubscriber<Integer> ts = Flowable.range(1, 5).test();

TestObserver<Integer> to = Observable.range(1, 5).test();

TestObserver<Integer> tso = Single.just(1).test();

TestObserver<Integer> tmo = Maybe.just(1).test();

TestObserver<Integer> tco = Completable.complete().test();
```

The second convenience is that most `TestSubscriber` / `TestObserver` methods return the instance itself allowing chaining the various `assertX` methods. The third convenience is that you can now fluently test your sources without the need to create or introduce `TestSubscriber` / `TestObserver` instance in your code:

```
Flowable.range(1, 5)
.test()
.assertResult(1, 2, 3, 4, 5)
;
```

Notable new assert methods

- `assertResult(T... items)`: asserts if subscribed, received exactly the given items in the given order followed by `onComplete` and no errors
- `assertFailure(Class<? extends Throwable> clazz, T... items)`: asserts if subscribed, received exactly the given items in the given order followed by a `Throwable` error of wich `clazz.isInstance()` returns true.
- `assertFailureAndMessage(Class<? extends Throwable> clazz, String message, T... items)`: same as `assertFailure` plus validates the `getMessage()` contains the specified message
- `awaitDone(long time, TimeUnit unit)` awaits a terminal event (blockingly) and cancels the sequence if the timeout elapsed.
- `assertOf(Consumer<TestSubscriber<T>> consumer)` compose some assertions into the fluent chain (used internally for fusion test as operator fusion is not part of the public API right now).

One of the benefits is that changing `Flowable` to `Observable` here the test code part doesn't have to change at all due to the implicit type change of the `TestSubscriber` to `TestObserver`.

cancel and request upfront

The `test()` method on `TestObserver` has a `test(boolean cancel)` overload which cancels/disposes the `TestSubscriber` / `TestObserver` before it even gets subscribed:

```java
PublishSubject<Integer> pp = PublishSubject.create();

// nobody subscribed yet
assertFalse(pp.hasSubscribers());

pp.test(true);

// nobody remained subscribed
assertFalse(pp.hasSubscribers());
```

`TestSubscriber` has the `test(long initialRequest)` and `test(long initialRequest, boolean cancel)` overloads to specify the initial request amount and whether the `TestSubscriber` should be also immediately cancelled. If the `initialRequest` is given, the `TestSubscriber` offers the `requestMore(long)` method to keep requesting in a fluent manner:

```java
Flowable.range(1, 5)
.test(0)
.assertValues()
.requestMore(1)
.assertValues(1)
.requestMore(2)
.assertValues(1, 2, 3)
.requestMore(2)
.assertResult(1, 2, 3, 4, 5);
```

or alternatively the `TestSubscriber` instance has to be captured to gain access to its `request()` method:

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

TestSubscriber<Integer> ts = pp.test(0L);

ts.request(1);

pp.onNext(1);
pp.onNext(2);

ts.assertFailure(MissingBackpressureException.class, 1);
```

Testing an async source

Given an asynchronous source, fluent blocking for a terminal event is now possible:

```java
Flowable.just(1)
.subscribeOn(Schedulers.single())
.test()
.awaitDone(5, TimeUnit.SECONDS)
.assertResult(1);
```

Mockito & TestSubscriber

Those who are using Mockito and mocked `Observer` in 1.x has to mock the `Subscriber.onSubscribe` method to issue an initial request, otherwise, the sequence will hang or fail with hot sources:

```
@SuppressWarnings("unchecked")
public static <T> Subscriber<T> mockSubscriber() {
    Subscriber<T> w = mock(Subscriber.class);

    Mockito.doAnswer(new Answer<Object>() {
        @Override
        public Object answer(InvocationOnMock a) throws Throwable {
            Subscription s = a.getArgumentAt(0, Subscription.class);
            s.request(Long.MAX_VALUE);
            return null;
        }
    }).when(w).onSubscribe((Subscription)any());

    return w;
}
```

Operator differences

Most operators are still there in 2.x and practically all of them have the same behavior as they had in 1.x. The following subsections list each base reactive type and the difference between 1.x and 2.x.

Generally, many operators gained overloads that now allow specifying the internal buffer size or prefetch amount they should run their upstream (or inner sources).

Some operator overloads have been renamed with a postfix, such as `fromArray`, `fromIterable` etc. The reason for this is that when the library is compiled with Java 8, the javac often can't disambiguate between functional interface types.

Operators marked as `@Beta` or `@Experimental` in 1.x are promoted to standard.

1.x Observable to 2.x Flowable

Factory methods:

1.x	2.x
`amb`	added `amb(ObservableSource...)` overload, 2-9 argument versions dropped
RxRingBuffer.SIZE	`bufferSize()`
`combineLatest`	added varargs overload, added overloads with `bufferSize` argument, `combineLatest(List)` dropped
`concat`	added overload with `prefetch` argument, 5-9 source overloads dropped, use `concatArray` instead

N/A	added `concatArray` and `concatArrayDelayError`
N/A	added `concatArrayEager` and `concatArrayEagerDelayError`
`concatDelayError`	added overloads with option to delay till the current ends or till the very end
`concatEagerDelayError`	added overloads with option to delay till the current ends or till the very end
`create(SyncOnSubscribe)`	replaced with `generate` + overloads (distinct interfaces, you can implement them all at once)
`create(AsnycOnSubscribe)`	not present
`create(OnSubscribe)`	repurposed with safe `create(FlowableOnSubscribe, BackpressureStrategy)`, raw support via `unsafeCreate()`
`from`	disambiguated into `fromArray`, `fromIterable`, `fromFuture`
N/A	added `fromPublisher`
`fromAsync`	renamed to `create()`
N/A	added `intervalRange()`
`limit`	dropped, use `take`
`merge`	added overloads with `prefetch`
`mergeDelayError`	added overloads with `prefetch`
`sequenceEqual`	added overload with `bufferSize`
`switchOnNext`	added overload with `prefetch`
`switchOnNextDelayError`	added overload with `prefetch`
`timer`	deprecated overloads dropped
`zip`	added overloads with `bufferSize` and `delayErrors` capabilities, disambiguated to `zipArray` and `zipIterable`

Instance methods:

1.x	2.x
`all`	**RC3** returns `Single<Boolean>` now
`any`	**RC3** returns `Single<Boolean>` now
`asObservable`	renamed to `hide()`, hides all identities now
`buffer`	overloads with custom `Collection` supplier
`cache(int)`	deprecated and dropped
`collect`	**RC3** returns `Single<U>`
`collect(U, Action2<U, T>)`	disambiguated to `collectInto` and **RC3** returns `Single<U>`

`concatMap`	added overloads with `prefetch`
`concatMapDelayError`	added overloads with `prefetch`, option to delay till the current ends or till the very end
`concatMapEager`	added overloads with `prefetch`
`concatMapEagerDelayError`	added overloads with `prefetch`, option to delay till the current ends or till the very end
`count`	**RC3** returns `Single<Long>` now
`countLong`	dropped, use `count`
`distinct`	overload with custom `Collection` supplier.
`doOnCompleted`	renamed to `doOnComplete`, note the missing `d`!
`doOnUnsubscribe`	renamed to `Flowable.doOnCancel` and `doOnDispose` for the others, [additional info](#)
N/A	added `doOnLifecylce` to handle `onSubscribe`, `request` and `cancel` peeking
`elementAt(int)`	**RC3** no longer signals NoSuchElementException if the source is shorter than the index
`elementAt(Func1, int)`	dropped, use `filter(predicate).elementAt(int)`
`elementAtOrDefault(int, T)`	renamed to `elementAt(int, T)` and **RC3** returns `Single<T>`
`elementAtOrDefault(Func1, int, T)`	dropped, use `filter(predicate).elementAt(int, T)`
`first()`	**RC3** renamed to `firstElement` and returns `Maybe<T>`
`first(Func1)`	dropped, use `filter(predicate).first()`
`firstOrDefault(T)`	renamed to `first(T)` and **RC3** returns `Single<T>`
`firstOrDefault(Func1, T)`	dropped, use `filter(predicate).first(T)`
`flatMap`	added overloads with `prefetch`
N/A	added `forEachWhile(Predicate<T>, [Consumer<Throwable>, [Action]])` for conditionally stopping consumption
`groupBy`	added overload with `bufferSize` and `delayError` option, *the custom internal map version didn't make it into RC1*
`ignoreElements`	**RC3** returns `Completable`
`isEmpty`	**RC3** returns `Single<Boolean>`
`last()`	**RC3** renamed to `lastElement` and returns `Maybe<T>`
`last(Func1)`	dropped, use `filter(predicate).last()`
`lastOrDefault(T)`	renamed to `last(T)` and **RC3** returns `Single<T>`

lastOrDefault(Func1, T)	dropped, use filter(predicate).last(T)
nest	dropped, use manual just
publish(Func1)	added overload with prefetch
reduce(Func2)	**RC3** returns Maybe<T>
N/A	added reduceWith(Callable, BiFunction) to reduce in a Subscriber-individual manner, returns Single<T>
N/A	added repeatUntil(BooleanSupplier)
repeatWhen(Func1, Scheduler)	dropped the overload, use subscribeOn(Scheduler).repeatWhen(Function) instead
retry	added retry(Predicate), retry(int, Predicate)
N/A	added retryUntil(BooleanSupplier)
retryWhen(Func1, Scheduler)	dropped the overload, use subscribeOn(Scheduler).retryWhen(Function) instead
sample	doesn't emit the very last item if the upstream completes within the period, added overloads with emitLast parameter
N/A	added scanWith(Callable, BiFunction) to scan in a Subscriber-individual manner
single()	**RC3** renamed to singleElement and returns Maybe<T>
single(Func1)	dropped, use filter(predicate).single()
singleOrDefault(T)	renamed to single(T) and **RC3** returns Single<T>
singleOrDefault(Func1, T)	dropped, use filter(predicate).single(T)
skipLast	added overloads with bufferSize and delayError options
startWith	2-9 argument version dropped, use startWithArray instead
N/A	added startWithArray to disambiguate
subscribe	No longer wraps all consumer types (i.e., Observer) with a safety wrapper, (just like the 1.x unsafeSubscribe no longer available). Use safeSubscribe to get an explicit safety wrapper around a consumer type.
N/A	added subscribeWith that returns its input after subscription
switchMap	added overload with prefetch argument
switchMapDelayError	added overload with prefetch argument
takeLastBuffer	dropped
N/A	added test() (returns TestSubscriber subscribed to this) with overloads to fluently test
throttleLast	doesn't emit the very last item if the upstream completes within the period,

	use `sample` with the `emitLast` parameter
`timeout(Func0<Observable>, ...)`	signature changed to `timeout(Publisher, ...)` and dropped the function, use `defer(Callable<Publisher>>)` if necessary
`toBlocking().y`	inlined as `blockingY()` operators, except `toFuture`
`toCompletable`	**RC3** dropped, use `ignoreElements`
`toList`	**RC3** returns `Single<List<T>>`
`toMap`	**RC3** returns `Single<Map<K, V>>`
`toMultimap`	**RC3** returns `Single<Map<K, Collection<V>>>`
N/A	added `toFuture`
N/A	added `toObservable`
`toSingle`	**RC3** dropped, use `single(T)`
`toSortedList`	**RC3** returns `Single<List<T>>`
`unsafeSubscribe`	Removed as the Reactive Streams specification mandates the `onXXX` methods don't crash and therefore the default is to not have a safety net in `subscribe`. The new `safeSubscribe` method was introduced to explicitly add the safety wrapper around a consumer type.
`withLatestFrom`	5-9 source overloads dropped
`zipWith`	added overloads with `prefetch` and `delayErrors` options

Different return types

Some operators that produced exactly one value or an error now return `Single` in 2.x (or `Maybe` if an empty source is allowed).

(Remark: this is "experimental" in RC2 and RC3 to see how it feels to program with such mixed-type sequences and whether or not there has to be too much `toObservable` / `toFlowable` back-conversion.)

Operator	Old return type	New return type	Remark
`all(Predicate)`	`Observable<Boolean>`	`Single<Boolean>`	Emits true if all elements match the predicate
`any(Predicate)`	`Observable<Boolean>`	`Single<Boolean>`	Emits true if any elements match the predicate
`count()`	`Observable<Long>`	`Single<Long>`	Counts the number of elements in the sequence
`elementAt(int)`	`Observable<T>`	`Maybe<T>`	Emits the element at the given index or completes
`elementAt(int, T)`	`Observable<T>`	`Single<T>`	Emits the element at the given index or the default

elementAtOrError(int)	Observable<T>	Single<T>	Emits the indexth element or a NoSuchElementException
first(T)	Observable<T>	Single<T>	Emits the very first element or NoSuchElementException
firstElement()	Observable<T>	Maybe<T>	Emits the very first element or completes
firstOrError()	Observable<T>	Single<T>	Emits the first element or a NoSuchElementException if the source is empty
ignoreElements()	Observable<T>	Completable	Ignore all but the terminal events
isEmpty()	Observable<Boolean>	Single<Boolean>	Emits true if the source is empty
last(T)	Observable<T>	Single<T>	Emits the very last element or the default item
lastElement()	Observable<T>	Maybe<T>	Emits the very last element or completes
lastOrError()	Observable<T>	Single<T>	Emits the lastelement or a NoSuchElementException if the source is empty
reduce(BiFunction)	Observable<T>	Maybe<T>	Emits the reduced value or completes
reduce(Callable, BiFunction)	Observable<U>	Single<U>	Emits the reduced value (or the initial value)
reduceWith(U, BiFunction)	Observable<U>	Single<U>	Emits the reduced value (or the initial value)
single(T)	Observable<T>	Single<T>	Emits the only element or the default item
singleElement()	Observable<T>	Maybe<T>	Emits the only element or completes
singleOrError()	Observable<T>	Single<T>	Emits the one and only element, IndexOutOfBoundsException if the source is longer than 1 item or a NoSuchElementException if the source is empty
toList()	Observable<List<T>>	Single<List<T>>	collects all elements into a List
toMap()	Observable<Map<K, V>>	Single<Map<K, V>>	collects all elements into a Map
toMultimap()	Observable<Map<K, Collection<V>>>	Single<Map<K, Collection<V>>>	collects all elements into a Map with collection
toSortedList()	Observable<List<T>>	Single<List<T>>	collects all elements into a List and sorts it

Removals

To make sure the final API of 2.0 is clean as possible, we remove methods and other components between release candidates without deprecating them.

Removed in version	Component	Remark
RC3	`Flowable.toCompletable()`	use `Flowable.ignoreElements()`
RC3	`Flowable.toSingle()`	use `Flowable.single(T)`
RC3	`Flowable.toMaybe()`	use `Flowable.singleElement()`
RC3	`Observable.toCompletable()`	use `Observable.ignoreElements()`
RC3	`Observable.toSingle()`	use `Observable.single(T)`
RC3	`Observable.toMaybe()`	use `Observable.singleElement()`

Miscellaneous changes

doOnCancel/doOnDispose/unsubscribeOn

In 1.x, the `doOnUnsubscribe` was always executed on a terminal event because 1.x' `SafeSubscriber` called `unsubscribe` on itself. This was practically unnecessary and the Reactive-Streams specification states that when a terminal event arrives at a `Subscriber`, the upstream `Subscription` should be considered cancelled and thus calling `cancel()` is a no-op.

For the same reason, `unsubscribeOn` is not called on the regular termination path but only when there is an actual `cancel` (or `dispose`) call on the chain.

Therefore, the following sequence won't call `doOnCancel`:

```
Flowable.just(1, 2, 3)
.doOnCancel(() -> System.out.println("Cancelled!"))
.subscribe(System.out::println);
```

However, the following will call since the `take` operator cancels after the set amount of `onNext` events have been delivered:

```
Flowable.just(1, 2, 3)
.doOnCancel(() -> System.out.println("Cancelled!"))
.take(2)
.subscribe(System.out::println);
```

If you need to perform cleanup on both regular termination or cancellation, consider the operator `using` instead.

Alternatively, the `doFinally` operator (introduced in 2.0.1 and standardized in 2.1) calls a developer specified `Action` that gets executed after a source completed, failed with an error or got cancelled/disposed:

```java
Flowable.just(1, 2, 3)
.doFinally(() -> System.out.println("Finally"))
.subscribe(System.out::println);

Flowable.just(1, 2, 3)
.doFinally(() -> System.out.println("Finally"))
.take(2) // cancels the above after 2 elements
.subscribe(System.out::println);
```