RxJava v3 Design

This document explains the terminology, principles, contracts, and other aspects of the design of RxJava v3. Its intended audience is the implementers of the library.

Terminology & Definitions

Interactive

Producer obeys consumer-driven flow control. Consumer manages capacity by requesting data.

Reactive

Producer is in charge. Consumer has to do whatever it needs to keep up.

Hot

When used to refer to a data source (such as an `Observable`), it means it does not have side-effects when subscribed to.

For example, an `Observable` of mouse events. Subscribing to that `Observable` does not cause the mouse events, but starts receiving them.

(Note: Yes, there are *some* side-effects of adding a listener, but they are inconsequential as far as the 'hot' usage is concerned).

Cold

When used to refer to a data source (such as an `Observable`), it means it has side-effects when subscribed to.

For example, an `Observable` of data from a remote API (such as an RPC call). Each time that `Observable` is subscribed to causes a new network call to occur.

Reactive/Push

Producer is in charge. Consumer has to do whatever it needs to keep up.

Examples:

- `Observable` (RxJS, Rx.Net, RxJava v1.x without backpressure, RxJava v2).
- Callbacks (the producer calls the function at its convenience).
- IRQ, mouse events, IO interrupts.
- 3.x `Flowable` (with `request(n)` credit always granted faster or in larger quantity than producer).
- Reactive Streams `Publisher` (with `request(n)` credit always granted faster or in larger quantity than producer).
- Java 9 `Flow.Publisher` (with `request(n)` credit always granted faster than or in larger quantity than producer).

Synchronous Interactive/Pull

Consumer is in charge. Producer has to do whatever it needs to keep up.

Examples:

- `Iterable`.
- 3.x/1.x `Observable` (without concurrency, producer and consumer on the same thread).
- 3.x `Flowable` (without concurrency, producer and consumer on the same thread).
- Reactive Streams `Publisher` (without concurrency, producer and consumer on the same thread).
- Java 9 `Flow.Publisher` (without concurrency, producer and consumer on the same thread).

Async Pull (Async Interactive)

Consumer requests data when it wishes, and the data is then pushed when the producer wishes to.

Examples:

- `Future` & `Promise`.
- `Single` (lazy `Future`).
- 3.x `Flowable`.
- Reactive Streams `Publisher`.
- Java 9 `Flow.Publisher`.
- 1.x `Observable` (with backpressure).
- `AsyncEnumerable` / `AsyncIterable`.

There is an overhead (performance and mental) for achieving this, which is why we also have the 3.x `Observable` without backpressure.

Flow Control

Flow control is any mitigation strategy that a consumer applies to reduce the flow of data.

Examples:

- Controlling the production of data, such as with `Iterator.next` or `Subscription.request(n)`.
- Preventing the delivery of data, such as buffer, drop, sample/throttle, and debounce.

Eager

Containing object immediately start work when it is created.

Examples:

- A `Future` once created has work being performed and represents the eventual value of that work. It can not be deferred once created.

Lazy

Containing object does nothing until it is subscribed to or otherwise started.

Examples:

- `Observable.create` does not start any work until `Observable.subscribe` starts the work.

RxJava & Related Types

Observable

Stream that supports async and synchronous push. It does *not* support interactive flow control (`request(n)`).

Usable for:

- Sync or async.
- Push.
- 0, 1, many or infinite items.

Flow control support:

- Buffering, sampling, throttling, windowing, dropping, etc.
- Temporal and count-based strategies.

Type Signature

```
class Observable<T> {
  void subscribe(Observer<T> observer);

  interface Observer<T> {
    void onNext(T t);
    void onError(Throwable t);
    void onComplete();
    void onSubscribe(Disposable d);
  }
}
```

The rule for using this type signature is:

> *onSubscribe onNext* (onError | onComplete)?*

Flowable

Stream that supports async and synchronous push and pull. It supports interactive flow control (`request(n)`).

Usable for:

- Pull sources.
- Push Observables with backpressure strategy (i.e. `Observable.toFlowable(onBackpressureStrategy)`).
- Sync or async.
- 0, 1, many or infinite items.

Flow control support:

- Buffering, sampling, throttling, windowing, dropping, etc.
- Temporal and count-based strategies.
- `request(n)` consumer demand signal:
  - For pull-based sources, this allows batched "async pull".
  - For push-based sources, this allows backpressure signals to conditionally apply strategies (i.e. drop, first, buffer, sample, fail, etc.).

You get a `Flowable` from:

- Converting a Observable with a backpressure strategy.
- Create from sync/async `onSubscribe` API (which participate in backpressure semantics).

Type Signature

```
class Flowable<T> implements Flow.Publisher<T>, io.reactivestreams.Publisher<T> {
  void subscribe(Subscriber<T> subscriber);

  interface Subscription implements Flow.Subscription,
io.reactivestreams.Subscription {
    void cancel();
    void request(long n);
  }
}
```

NOTE: To support `Flow.Publisher` in Java 9+ without breaking Java 7+ compatibility, we want to use the multi-release jar file support.

The rule for using this type signature is:

> *onSubscribe onNext* (onError | onComplete)?*

Single

Lazy representation of a single response (lazy equivalent of `Future` / `Promise`).

Usable for:

- Pull sources.
- Push sources being windowed or flow controlled (such as `window(1)` or `take(1)`).
- Sync or async.
- 1 item.

Flow control:

- Not applicable (don't subscribe if the single response is not wanted).

Type Signature

```
class Single<T> {
  void subscribe(Single.Subscriber<T> subscriber);
}

interface SingleSubscriber<T> {
  void onSuccess(T t);
  void onError(Throwable t);
  void onSubscribe(Disposable d);
}
```

> *onSubscribe (onError | onSuccess)?*

Completable

Lazy representation of a unit of work that can complete or fail.

- Semantic equivalent of `Observable.empty().doOnSubscribe()`.
- Alternative for scenarios often represented with types such as `Single<Void>` or `Observable<Void>`.

Usable for:

- Sync or async.
- 0 items.

Type Signature

```
class Completable {
  void subscribe(Completable.Subscriber subscriber);
}

interface CompletableSubscriber {
  void onComplete();
  void onError(Throwable t);
```

```
  void onSubscribe(Disposable d);
}
```

> *onSubscribe (onError | onComplete)?*

Observer

Reactive consumer of events (without consumer-driven flow control). Involved in subscription lifecycle to allow unsubscription.

Publisher

Interactive producer of events (with flow control). Implemented by `Flowable`.

Reactive Streams producer of data.

Subscriber

Interactive consumer of events (with consumer-driven flow control). Involved in subscription lifecycle to allow unsubscription.

Reactive Streams consumer of data.

Subscription

Reactive Streams state of subscription supporting flow control and cancellation.

`Disposable` is a similar type used for lifecycle management on the `Observable` type without interactive flow control.

Processor

Reactive Streams operator for defining behavior between `Publisher` and `Subscriber`. It must obey the contracts of `Publisher` and `Subscriber`, meaning it is sequential, serialized, and must obey `request(n)` flow control.

Subject

A "hot", push-based data source that allows a producer to emit events to it and consumers to subscribe to events in a multicast manner. It is "hot" because consumers subscribing to it does not cause side-effects, or affect the data flow in any way. It is push-based and reactive because the producer is fully in charge.

A `Subject` is used to decouple unsubscription. Termination is fully in the control of the producer. `onError` and `onComplete` are still terminal events. `Subject`s are stateful and retain their terminal state (for replaying to all/future subscribers).

Relation to Reactive Streams:

- It can not implement Reactive Streams `Publisher` unless it is created with a default consumer-driven flow control strategy.
- It can not implement `Processor` since a `Processor` must compose `request(n)` which can not be done with multicasting or push.

Here is an approach to converting from a `Subject` to Reactive Streams types by adding a default flow control strategy:

```
Subject s = PublishSubject.create();
// convert to Publisher with backpressure strategy
Publisher p = s.toPublisher(onBackpressureStrategy);
```

```
// now the request(n) semantics are handled by default
p.subscribe(subscriber1);
p.subscribe(subscriber2);
```

In this example, `subscriber1` and `subscriber2` can consume at different rates, `request(n)` will propagate to the provided `onBackpressureStrategy`, not the original `Subject` which can't propagate `request(n)` upstream.

Disposable

A type representing work or resource that can be cancelled or disposed.

Examples:

- An `Observable.subscribe` passes a `Disposable` to the `Observable.onSubscribe` to allow the `Observer` to dispose of the subscription.
- A `Scheduler` returns a `Disposable` that you use for disposing of the `Scheduler`.

`Subscription` is a similar type used for lifecycle management on the `Flowable` type with interactive flow control.

Operator

An operator follows a specific lifecycle (union of the producer/consumer contract).

- It must propagate the `subscribe` event upstream (to the producer).
- It must obey the RxJava contract (serialize all events, `onError` / `onComplete` are terminal).
- If it has resources to cleanup it is responsible for watching `onError`, `onComplete`, and `cancel/dispose`, and doing the necessary cleanup.
- It must propagate the `cancel/dispose` upstream.

In the addition of the previous rules, an operator for `Flowable`:

- It must propagate/negotiate the `request(n)` event.

Creation

Unlike RxJava 1.x, 3.x base classes are to be abstract, stateless and generally no longer wrap an `onSubscribe` callback - this saves allocation in assembly time without limiting the expressiveness. Operator methods and standard factories still live as final on the base classes.

Instead of the indirection of an `onSubscribe` and `lift`, operators are to be implemented by extending the base classes. For example, the `map` operator will look like this:

```
public final class FlowableMap<T, R> extends Flowable<R> {

    final Flowable<? extends T> source;

    final Function<? super T, ? extends R> mapper;

    public FlowableMap(Flowable<? extends T> source, Function<? super T, ? extends
R> mapper) {
        this.source = source;
        this.mapper = mapper;
```

```
    }

    @Override
    protected void subscribeActual(Subscriber<? super R> subscriber) {
        source.subscribe(new FlowableMapSubscriber<T, R>(subscriber, mapper));
    }

    static final class FlowableMapSubscriber<T, R> implements Subscriber<T>,
Subscription {
        // ...
    }
}
```

Since Java still doesn't have extension methods, "adding" more operators can only happen through helper methods such as `lift(C -> C)` and `compose(R -> P)` where `C` is the default consumer type (i.e. `rs.Subscriber`), `R` is the base type (i.e. `Flowable`) and `P` is the base interface (i.e. `rs.Publisher`). As before, the library itself may gain or lose standard operators and/or overloads through the same community process.

In concert, `create(onSubscribe)` will not be available; standard operators extend the base types directly. The conversion of other RS-based libraries will happen through the `Flowable.wrap(Publisher<T>)` static method.

(*The unfortunate effect of `create` in 1.x was the ignorance of the Observable contract and beginner's first choice as an entry point. We can't eliminate this path since `rs.Publisher` is a single method functional interface that can be implemented just as badly.*)

Therefore, new standard factory methods will try to address the common entry point requirements.

The `Flowable` will contain the following `create` methods:

- `create(SyncGenerator<T, S>)`: safe, synchronous generation of signals, one-by-one.
- `create(AsyncOnSubscribe<T, S>)`: batch-create signals based on request patterns.
- `create(Consumer<? super FlowEmitter<T>>)`: relay multiple values or error from multi-valued reactive-sources (i.e. button-clicks) while also give flow control options right there (buffer, drop, error, etc.).
- `createSingle(Consumer<? super SingleEmitter<T>>)`: relay a single value or error from other reactive sources (i.e. addListener callbacks).
- `createEmpty(Consumer<? super CompletionEmitter>)`: signal a completion or error from valueless reactive sources.

The `Observable` will contain the following `create` methods:

- `create(SyncGenerator<T, S>)`: safe, synchronous generation of signals, one-by-one.
- `create(Consumer<? super FlowEmitter<T>>)`: relay multiple values or error from multi-valued reactive-sources (i.e. button-clicks) while also give flow control options right there (buffer, drop, error, etc.).
- `createSingle(Consumer<? super SingleEmitter<T>>)`: relay a single value or error from other reactive sources (i.e. addListener callbacks).
- `createEmpty(Consumer<? super CompletionEmitter>)`: signal a completion or error from valueless reactive sources.

The `Single` will contain the following `create` method:

- `create(Consumer<? super SingleEmitter<T>>)`: relay a single value or error from other reactive sources (i.e. addListener callbacks).

The `Completable` will contain the following `create` method:

- `create(Consumer<? super CompletionEmitter>)` : signal a completion or error from valueless reactive sources.

The first two `create` methods take an implementation of an interface which provides state and the generator methods:

```
interface SyncGenerator<T, S> {

    S createState();

    S generate(S state, Observer<T> output);

    void disposeState(S state);
}

interface AsyncGenerator<T, S> {

    S createState();

    S generate(S state, long requested, Observer<Observable<T>> output);

    void disposeState(S state);
}
```

These latter three `create` methods will provide the following interaction interfaces to the `java.util.function.Consumer` :

```
interface SingleEmitter<T> {

    complete(T value);

    fail(Throwable error);

    stop();

    setDisposable(Disposable d);

}

interface FlowEmitter<T> {

    void next(T value);

    void fail(Throwable error);

    void complete();

    void stop();
```

```
    setDisposable(Disposable d);

    enum BackpressureHandling {
        IGNORE,
        ERROR,
        DROP,
        LATEST,
        BUFFER
    }

    void setBackpressureHandling(BackpressureHandling mode);

}

interface CompletableEmitter<T> {

    complete();

    fail(Throwable error);

    stop();

    setDisposable(Disposable d);

}
```

By extending the base classes, operator implementations would loose the tracking/wrapping features of 1.x. To avoid this, the methods `subscribe(C)` will be final and operators have to implement a protected `subscribeActual` (or any other reasonable name).

```
@Override
public final void subscribe(Subscriber<? super T> s) {
   subscribeActual(hook.onSubscribe(s));
}

protected abstract void subscribeActual(Subscriber<? super T> s);
```

Assembly-time hooks will be moved into the individual standard methods on the base types:

```
public final Flowable<R> map(Function<? super T, ? extends R> mapper) {
    return hook.onAssembly(new FlowableMap<T, R>(this, mapper));
}
```

Terminal behavior

A producer can terminate a stream by emitting `onComplete` or `onError`. A consumer can terminate a stream by calling `cancel` / `dispose`.

Any resource cleanup of the source or operators must account for any of these three termination events. In other words, if an operator needs cleanup, then it should register the cleanup callback with `cancel` / `dispose`,

`onError` and `onComplete`.

The final `subscribe` will *not* invoke `cancel` / `dispose` after receiving an `onComplete` or `onError`.

JVM target and source compatibility

The 3.x version will target JDK6+ to let Android users consume the new version of RxJava.

Future work

This section contains current design work which needs more discussion and elaboration before it is merged into this document as a stated goal for 3.x.

Custom Observable, Single, Completable, or Flowable

We are investigate a base interface (similar to `Publisher`) for the `Observable`, `Single`, and `Completable` (currently referred to as `Consumable` or `ConsumableObservable`). This would empower library owners and api developers to implement their own type of `Observable`, `Single`, or `Completable` without extending the class. This would result in a change the type signatures of `subscribe` as well as any operator that operates over an `Observable`, `Single`, or `Completable` to accept a more generic type (i.e. `ConsumableObservable`). For more information see the proof of concept project Consumable.

Fusion

Operator fusion exploits the declarative nature of building flows; the developer specifies the "what", "where" and "when", the library then tries to optimize the "how".

There are two main levels of operator fusion: *macro* and *micro*.

Macro-fusion

Macro fusion deals with the higher level view of the operators, their identity and their combination (mostly in the form of subsequence). This is partially an internal affair of the operators, triggered by the downstream operator and may work with several cases. Given an operator application pair `a().b()` where `a` could be a source or an intermediate operator itself, when the application of `b` happens in assembly time, the following can happen:

- `b` identifies `a` and decides to not apply itself. Example: `empty().flatMap()` is functionally a no-op.
- `b` identifies `a` and decides to apply a different, conventional operator. Example: `just().subscribeOn()` is turned into `just().observeOn()`.
- `b` decides to apply a new custom operator, combining and inlining existing behavior. Example: `just().subscribeOn()` internally goes to `ScalarScheduledPublisher`.
- `a` is `b` and the two operator's parameter set can be combined into a single application. Example: `filter(p1).filter(p2)` combined into `filter(p1 && p2)`.

Participating in the macro-fusion externally is possible by implementing a marker interface when extending `Flowable`. Two kinds of interfaces are available:

- `java.util.Callable`: the Java standard, throwing interface, indicating the single value has to be extracted in subscription time (or later).
- `ScalarCallable`: to indicate the single value can be safely extracted during assembly time and used/inlined in other operators:

```
interface ScalarCallable<T> extends java.util.Callable<T> {
    @Override
```

```
    T call();
}
```

`ScalarCallable` is also `Callable` and thus its value can be extracted practically anytime. For convenience (and for sense), `ScalarCallable` overrides and hides the superclass' `throws Exception` clause - throwing during assembly time is likely unreasonable for scalars.

Since Reactive-Streams doesn't allow `null` s in the value flow, we have the opportunity to define `ScalarCallable` s and `Callable` s returning `null` should be considered as an empty source - allowing operators to dispatch on the type `Callable` first then branch on the nullness of `call()` .

Interoperating with other libraries, at this level is possible. Reactor-Core uses the same pattern and the two libraries can work with each other's `Publisher+Callable` types. Unfortunately, this means subscription-time only fusion as `ScalarCallable` s live locally in each library.

Micro-fusion

Micro-fusion goes a step deeper and tries to reuse internal structures, mostly queues, in operator pairs, saving on allocation and sometimes on atomic operations. It's property is that, in a way, subverts the standard Reactive-Streams protocol between subsequent operators that both support fusion. However, from the outside world's view, they still work according to the RS protocol.

Currently, two main kinds of micro-fusion opportunities are available.

1) CONDITIONAL SUBSCRIBER

This extends the RS `Subscriber` interface with an extra method: `boolean tryOnNext(T value)` and can help avoiding small request amounts in case an operator didn't forward but dropped the value. The canonical use is for the `filter()` operator where if the predicate returns false, the operator has to request 1 from upstream (since the downstream doesn't know there was a value dropped and thus not request itself). Operators wanting to participate in this fusion have to implement and subscribe with an extended `Subscriber` interface:

```
interface ConditionalSubscriber<T> {
   boolean tryOnNext(T value);
}

//...
@Override
protected void subscribeActual(Subscriber<? super T> s) {
    if (s instanceof ConditionalSubscriber) {
        source.subscribe(new FilterConditionalSubscriber<>(s, predicate));
    } else {
        source.subscribe(new FilterRegularSubscriber<>(s, predicate));
    }
}
```

(Note that this may lead to extra case-implementations in operators that have some kind of queue-drain emission model.)

2) QUEUE-FUSION

The second category is when two (or more) operators share the same underlying queue and each append activity at the exit point (i.e. `poll()`) of the queue. This can work in two modes: synchronous and asynchronous.

In synchronous mode, the elements of the sequence is already available (i.e. a fixed `range()` or `fromArray()`, or can be synchronously calculated in a pull fashion in `fromIterable`. In this mode, the requesting and regular onError-path is bypassed and is forbidden. Sources have to return null from `pull()` and false from `isEmpty()` if they have no more values and throw from these methods if they want to indicate an exceptional case.

In asynchronous mode, elements may become available at any time, therefore, `pull` returning null, as with regular queue-drain, is just the indication of temporary lack of source values. Completion and error still has to go through `onComplete` and `onError` as usual, requesting still happens as usual but when a value is available in the shared queue, it is indicated by an `onNext(null)` call. This can trigger a chain of `drain` calls without moving values in or out of different queues.

In both modes, `cancel` works and behaves as usual.

Since this fusion mode is an optional extension, the mode switch has to be negotiated and the shared queue interface established. Operators already working with internal queues then can, mostly, keep their current `drain()` algorithm. Queue-fusion has its own interface and protocol built on top of the existing `onSubscribe` - `Subscription` rail:

```
interface QueueSubscription<T> implements Queue<T>, Subscription {
    int NONE = 0;
    int SYNC = 1;
    int ASYNC = 2;
    int ANY = SYNC | ASYNC;
    int BOUNDARY = 4;

    int requestFusion(int mode);
}
```

For performance, the mode is an integer bitflags setup, called early during subscription time, and allows negotiating the fusion mode. Usually, producers can do only one mode and consumers can do both mode. Because fused, intermediate operators attach logic (which is many times user-callback) to the exit point of the queue interface (poll()), it may change the computation location of those callbacks in an unwanted way. The flag `BOUNDARY` is added by consumers indicating that they will consume the queue over an async boundary. Intermediate operators, such as `map` and `filter` then can reject the fusion in such sequences.

Since RxJava 3.x is still JDK 6 compatible, the `QueueSubscription` can't itself default unnecessary methods and implementations are required to throw `UnsupportedOperationException` for `Queue` methods other than the following:

- `poll()`.
- `isEmpty()`.
- `clear()`.
- `size()`.

Even though other modern libraries also define this interface, they live in local packages and thus non-reusable without dragging in the whole library. Therefore, until externalized and standardized, cross-library micro-fusion won't happen.

A consequence of the extension of the `onSubscribe` - `Subscription` rail is that intermediate operators are no longer allowed to pass an upstream `Subscription` directly to its downstream `Subscriber.onSubscribe`. Doing so is likely to have the fused sequence skip the operator completely, losing behavior or causing runtime

exceptions. Since RS `Subscriber` is an interface, operators can simply implement both `Subscriber` and `Subscription` on themselves, delegating the `request` and `cancel` calls to the upstream and calling `child.onSubscribe(this)`.