Table of contents

Introduction

Writing operators, source-like (`fromEmitter`) or intermediate-like (`flatMap`) **has always been a hard task to do in RxJava**. There are many rules to obey, many cases to consider but at the same time, many (legal) shortcuts to take to build a well performing code. Now writing an operator specifically for 2.x is 10 times harder than for 1.x. If you want to exploit all the advanced, 4th generation features, that's even 2-3 times harder on top (so 30 times harder in total).

(If you have been following my blog about RxJava internals, writing operators

is maybe only 2 times harder than 1.x; some things have moved around, some tools popped up while others have been dropped but there is a relatively straight mapping from 1.x concepts and approaches to 2.x concepts and approaches.)

In this article, I'll describe the how-to's from the perspective of a developer who skipped the 1.x knowledge base and basically wants to write operators that conforms the Reactive-Streams specification as well as RxJava 2.x's own extensions and additional expectations/requirements.

Since **Reactor 3** has the same architecture as **RxJava 2** (no accident, I architected and contributed 80% of **Reactor 3** as well) the same principles outlined in this page applies to writing operators for **Reactor 3**. Note however that they chose different naming and locations for their utility and support classes so you may have to search for the equivalent components.

Warning on internal components

RxJava has several hundred public classes implementing various operators and helper facilities. Since there is no way to hide these in Java 6-8, the general contract is that anything below `io.reactivex.internal` is considered private and subject to change without warnings. It is not recommended to reference these in your code (unless you contribute to RxJava itself) and must be prepared that even a patch change may shuffle/rename things around in them. That being said, they usually contain valuable tools for operator builders and as such are quite attractive to use them in your custom code.

Atomics, serialization, deferred actions

As RxJava itself has building blocks for creating reactive dataflows, its components have building blocks as well in the form of concurrency primitives and algorithms. Many refer to the book Concurrency in Practice for learning the fundamentals needed. Unfortunately, other than some explanation of the Java Memory Model, the book lacks the techniques required for developing operators for RxJava 1.x and 2.x.

Field updaters and Android

If you looked at the source code of RxJava and then at **Reactor 3**, you might have noticed that RxJava doesn't use the `AtomicXFieldUpdater` classes. The reason for this is that on certain Android devices, the runtime "messes up" the field names and the reflection logic behind the field updaters fails to locate those fields in the operators. To avoid this we decided to only use the full `AtomicX` classes (as fields or extending them).

If you target the RxJava library with your custom operator (or Android), you are encouraged to do the same. If you plan have operators running on desktop Java, feel free to use the field updaters instead.

Request accounting

When dealing with backpressure in `Flowable` operators, one needs a way to account the downstream requests and emissions in response to those requests. For this we use a single `AtomicLong`. Accounting must be atomic because requesting more and emitting items to fulfill an earlier request may happen at the same time.

The naive approach for accounting would be to simply call `AtomicLong.getAndAdd()` with new requests and `AtomicLong.addAndGet()` for decrementing based on how many elements were emitted.

The problem with this is that the Reactive-Streams specification declares `Long.MAX_VALUE` as the upper bound for outstanding requests (interprets it as the unbounded mode) but adding two large longs may overflow the `long` into a negative value. In addition, if for some reason, there are more values emitted than were requested, the subtraction may yield a negative current request value, causing crashes or hangs.

Therefore, both addition and subtraction have to be capped at `Long.MAX_VALUE` and `0` respectively. Since there is no dedicated `AtomicLong` method for it, we have to use a Compare-And-Set loop. (Usually, requesting happens relatively rarely compared to emission amounts thus the lack of dedicated machine code instruction is not a performance bottleneck.)

```
public static long add(AtomicLong requested, long n) {
    for (;;) {
        long current = requested.get();
        if (current == Long.MAX_VALUE) {
            return Long.MAX_VALUE;
        }
        long update = current + n;
        if (update < 0L) {
            update = Long.MAX_VALUE;
        }
        if (requested.compareAndSet(current, update)) {
            return current;
        }
    }
}

public static long produced(AtomicLong requested, long n) {
for (;;) {
        long current = requested.get();
        if (current == Long.MAX_VALUE) {
            return Long.MAX_VALUE;
        }
        long update = current - n;
```

```
            if (update < 0L) {
                update = 0;
            }
            if (requested.compareAndSet(current, update)) {
                return update;
            }
        }
}
```

In fact, these are so common in RxJava's operators that these algorithms are available as utility methods on the **internal** `BackpressureHelper` class under the same name.

Sometimes, instead of having a separate `AtomicLong` field, your operator can extend `AtomicLong` saving on the indirection and class size. The practice in RxJava 2.x operators is that unless there is another atomic counter needed by the operator, (such as work-in-progress counter, see the later subsection) and otherwise doesn't need a base class, they extend `AtomicLong` directly.

The `BackpressureHelper` class features special versions of `add` and `produced` which treat `Long.MIN_VALUE` as a cancellation indication and won't change the `AtomicLong`s value if they see it.

Once

RxJava 2 expanded the single-event reactive types to include `Maybe` (called as the reactive `Optional` by some). The common property of `Single`, `Completable` and `Maybe` is that they can only call one of the 3 kinds of methods on their consumers: `(onSuccess | onError | onComplete)`. Since they also participate in concurrent scenarios, an operator needs a way to ensure that only one of them is called even though the input sources may call multiple of them at once.

To ensure this, operators may use the `AtomicBoolean.compareAndSet` to atomically chose the event to relay (and thus the other events to drop).

```
final AtomicBoolean once = new AtomicBoolean();

final MaybeObserver<? super T> child = ...;

void emitSuccess(T value) {
   if (once.compareAndSet(false, true)) {
       child.onSuccess(value);
   }
}

void emitFailure(Throwable e) {
    if (once.compareAndSet(false, true)) {
        child.onError(e);
    } else {
        RxJavaPlugins.onError(e);
    }
}
```

Note that the same sequential requirement applies to these 0-1 reactive sources as to `Flowable`/`Observable`, therefore, if your operator doesn't have to deal with events from multiple sources (and pick one of them), you don't need this construct.

Serialization

With more complicated sources, it may happen that multiple things happen that may trigger emission towards the downstream, such as upstream becoming available while the downstream requests for more data while the sequence gets cancelled by a timeout.

Instead of working out the often very complicated state transitions via atomics, perhaps the easiest way is to serialize the events, actions or tasks and have one thread perform the necessary steps after that. This is what I call **queue-drain** approach (or trampolining by some).

(The other approach, **emitter-loop** is no longer recommended with 2.x due to its potential blocking `synchronized` constructs that looks performant in single-threaded case but destroys it in true concurrent case.)

The concept is relatively simple: have a concurrent queue and a work-in-progress atomic counter, enqueue the item, increment the counter and if the counter transitioned from 0 to 1, keep draining the queue, work with the element and decrement the counter until it reaches zero again:

```
final ConcurrentLinkedQueue<Runnable> queue = ...;
final AtomicInteger wip = ...;

void execute(Runnable r) {
    queue.offer(r);
    if (wip.getAndIncrement() == 0) {
        do {
            queue.poll().run();
        } while (wip.decrementAndGet() != 0);
    }
}
```

The same pattern applies when one has to emit onNext values to a downstream consumer:

```
final ConcurrentLinkedQueue<T> queue = ...;
final AtomicInteger wip = ...;
final Subscriber<? super T> child = ...;
```

```
void emit(T r) {
    queue.offer(r);
    if (wip.getAndIncrement() == 0) {
        do {
            child.onNext(queue.poll());
        } while (wip.decrementAndGet() != 0);
    }
}
```

Queues

Using `ConcurrentLinkedQueue` is a reliable although mostly an overkill for such situations because it allocates on each call to `offer()` and is unbounded. It can be replaced with more optimized queues (see JCTools) and RxJava itself also has some customized queues available (internal!):

- `SpscArrayQueue` used when the queue is known to be fed by a single thread but the serialization has to look at other things (request, cancellation, termination) that can be read from other fields. Example: `observeOn` has a fixed request pattern which fits into this type of queue and extra fields for passing an error, completion or downstream requests into the drain logic.
- `SpscLinkedArrayQueue` used when the queue is known to be fed by a single thread but there is no bound on the element count. Example: `UnicastProcessor`, almost all buffering `Observable` operator. Some operators use it with multiple event sources by synchronizing on the `offer` side - a tradeoff between allocation and potential blocking:

```
SpscLinkedArrayQueue<T> q = ...
synchronized(q) {
    q.offer(value);
}
```

- `MpscLinkedQueue` where there could be many feeders and unknown number of elements. Example: `buffer` with reactive boundary.

The RxJava 2.x implementations of these types of queues have different class hierarchy than the JDK/JCTools versions. Our classes don't implement the `java.util.Queue` interface but rather a custom, simplified interface:

```
interface SimpleQueue<T> {
    boolean offer(T t);

    boolean offer(T t1, T t2);

    T poll() throws Exception;
    boolean isEmpty();

    void clear();
}

interface SimplePlainQueue<T> extends SimpleQueue<T> {
    @Override
    T poll();
}

public final class SpscArrayQueue<T> implements SimplePlainQueue<T> {
    // ...
}
```

This simplified queue API gets rid of the unused parts (iterator, collections API remnants) and adds a bi-offer method (only implemented atomically in `SpscLinkedArrayQueue` currently). The second interface, `SimplePlainQueue` is defined to suppress the `throws Exception` on poll on queue types that won't throw that exception and there is no need for try-catch around them.

Deferred actions

The Reactive-Streams has a strict requirement that calling `onSubscribe()` must happen before any calls to the rest of the `onXXX` methods and by nature, any calls to `Subscription.request()` and `Subscription.cancel()`. The same logic applies to the design of `Observable`, `Single`, `Completable` and `Maybe` with their connection type of `Disposable`.

Often though, such call to `onSubscribe` may happen later than the respective `cancel()` needs to happen. For example, the user may want to call `cancel()` before the respective `Subscription` actually becomes available in `subscribeOn`. Other operators may need to call `onSubscribe` before they connect to other sources but at that time, there is no direct way for relaying a `cancel` call to an unavailable upstream `Subscription`.

The solution is **deferred cancellation** and **deferred requesting** in general.

Deferred cancellation

This approach affects all 5 reactive types and works the same way for everyone. First, have an `AtomicReference` that will hold the respective connection type (or any other type whose method call has to happen later). Two methods are needed handling the `AtomicReference` class, one that sets the actual instance and one that calls the `cancel`/`dispose` method on it.

```
static final Disposable DISPOSED;
static {
    DISPOSED = Disposables.empty();
    DISPOSED.dispose();
}

static boolean set(AtomicReference<Disposable> target, Disposable value) {
    for (;;) {
        Disposable current = target.get();
        if (current == DISPOSED) {
            if (value != null) {
                value.dispose();
            }
            return false;
        }
        if (target.compareAndSet(current, value)) {
            if (current != null) {
                current.dispose();
            }
            return true;
        }
    }
}

static boolean dispose(AtomicReference<Disposable> target) {
    Disposable current = target.getAndSet(DISPOSED);
    if (current != DISPOSED) {
        if (current != null) {
            current.dispose();
        }
        return true;
    }
    return false;
}
```

The approach uses an unique sentinel value `DISPOSED` - that should not appear elsewhere in your code - to indicate once a late actual `Disposable` arrives, it should be disposed immediately. Both methods return true if the operation succeeded or false if the target was already disposed.

Sometimes, only one call to `set` is permitted (i.e., `setOnce`) and other times, the previous non-null value needs no call to `dispose` because it is known to be disposed already (i.e., `replace`).

As with the request management, there are utility classes and methods for these operations:

- (internal) `SequentialDisposable` that uses `update`, `replace` and `dispose` but leaks the API of `AtomicReference`
- `SerialDisposable` that has safe API with `set`, `replace` and `dispose` among other things

- (internal) `DisposableHelper` that features the methods shown above and the global disposed sentinel used by RxJava. It may come handy when one uses `AtomicReference<Disposable>` as a base class.

The same pattern applies to `Subscription` with its `cancel()` method and with helper (internal) class `SubscriptionHelper` (but no `SequentialSubscription` or `SerialSubscription`, see next subsection).

Deferred requesting

With `Flowables` (and Reactive-Streams `Publishers`) the `request()` calls need to be deferred as well. In one form (the simpler one), the respective late `Subscription` will eventually arrive and we need to relay all previous and all subsequent request amount to its `request()` method.

In 1.x, this behavior was implicitly provided by `rx.Subscriber` but at a high cost that had to be payed by all instances whether or not they needed this feature.

The solution works by having the `AtomicReference` for the `Subscription` and an `AtomicLong` to store and accumulate the requests until the actual `Subscription` arrives, then atomically request all deferred value once.

```java
static boolean deferredSetOnce(AtomicReference<Subscription> subscription,
        AtomicLong requested, Subscription newSubscription) {
    if (subscription.compareAndSet(null, newSubscription) {
        long r = requested.getAndSet(0L);
        if (r != 0) {
            newSubscription.request(r);
        }
        return true;
    }
    newSubscription.cancel();
    if (subscription.get() != SubscriptionHelper.CANCELLED) {
        RxJavaPlugins.onError(new IllegalStateException("Subscription already set!"));
    }
    return false;
}

static void deferredRequest(AtomicReference<Subscription> subscription,
        AtomicLong requested, long n) {
    Subscription current = subscription.get();
    if (current != null) {
        current.request(n);
    } else {
        BackpressureHelper.add(requested, n);
        current = subscription.get();
        if (current != null) {
            long r = requested.getAndSet(0L);
            if (r != 0L) {
                current.request(r);
            }
        }
    }
}
```

In deferredSetOnce, if the CAS from null to the newSubscription succeeds, we atomically exchange the request amount to 0L and if the original value was nonzero, we request from newSubscription. In deferredRequest, if there is a Subscription we simply request from it directly. Otherwise, we accumulate the requests via the helper method then check again if the Subscription arrived or not. If it arrived in the meantime, we atomically exchange the accumulated request value and if nonzero, request it from the newly retrieved Subscription. This non-blocking logic makes sure that in case of concurrent invocations of the two methods, no accumulated request is left behind.

This complex logic and methods, along with other safeguards are available in the (internal) SubscriptionHelper utility class and can be used like this:

```
final class Operator<T> implements Subscriber<T>, Subscription {
    final Subscriber<? super T> child;

    final AtomicReference<Subscription> ref = new AtomicReference<>();
    final AtomicLong requested = new AtomicLong();

    public Operator(Subscriber<? super T> child) {
        this.child = child;
    }

    @Override
    public void onSubscribe(Subscription s) {
        SubscriptionHelper.deferredSetOnce(ref, requested, s);
    }

    @Override
    public void onNext(T t) { ... }

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

    @Override
    public void onComplete() { ... }

    @Override
    public void cancel() {
        SubscriptionHelper.cancel(ref);
    }

    @Override
    public void request(long n) {
        SubscriptionHelper.deferredRequested(ref, requested, n);
    }
}

Operator<T> parent = new Operator<T>(child);

child.onSubscribe(parent);

source.subscribe(parent);
```

The second form is when multiple `Subscription`s replace each other and we not only need to hold onto request amounts when there is none of them but make sure a newer `Subscription` is requested only that much the previous `Subscription`'s upstream didn't deliver. This is called **Subscription arbitration** and the relevant algorithms are quite verbose and will be omitted here. There is, however, an utility class that manages this: (internal) `SubscriptionArbiter`.

You can extend it (to save on object headers) or have it as a field. Its main use is to send it to the downstream via `onSubscribe` and update its current `Subscription` in the current operator. Note that even though its methods are thread-safe, it is intended for swapping `Subscription`s when the current one finished emitting events. This makes sure that any newer `Subscription` is requested the right amount and not more due to production/switch race.

```
final SubscriptionArbiter arbiter = ...

// ...

child.onSubscribe(arbiter);

// ...

long produced;

@Override
public void onSubscribe(Subscription s) {
    arbiter.setSubscription(s);
}

@Override
public void onNext(T value) {
   produced++;
```

```
    child.onNext(value);
}

@Override
public void onComplete() {
    long p = produced;
    if (p != 0L) {
        arbiter.produced(p);
    }
    subscribeNext();
}
```

For better performance, most operators can count the produced element amount and issue a single `SubscriptionArbiter.produced()` call just before switching to the next `Subscription`.

Atomic error management

In some cases, multiple sources may signal a `Throwable` at the same time but the contract forbids calling `onError` multiple times. Once can, of course use the **once** approach with `AtomicReference<Throwable>` and throw out but the first one to set the `Throwable` on it.

The alternative is to collect these `Throwable`s into a `CompositeException` as long as possible and at one point lock out the others. This works by doing a copy-on-write scheme or by linking `CompositeException`s atomically and having a terminal sentinel to indicate all further errors should be dropped.

```
static final Throwable TERMINATED = new Throwable();

static boolean addThrowable(AtomicReference<Throwable> ref, Throwable e) {
    for (;;) {
        Throwable current = ref.get();
        if (current == TERMINATED) {
            return false;
        }
        Throwable next;
        if (current == null) {
            next = e;
        } else {
            next = new CompositeException(current, e);
        }
        if (ref.compareAndSet(current, next)) {
            return true;
        }
    }
}
```

```java
static Throwable terminate(AtomicReference<Throwable> ref) {
    return ref.getAndSet(TERMINATED);
}
```

as with most common logic, this is supported by the (internal) `ExceptionHelper` utility class and the (internal) `AtomicThrowable` class.

The usage pattern looks as follows:

```java
final AtomicThrowable errors = ...;

@Override
public void onError(Throwable e) {
    if (errors.addThrowable(e)) {
        drain();
    } else {
        RxJavaPlugins.onError(e);
    }
}

void drain() {
   // ...
   if (errors.get() != null) {
       child.onError(errors.terminate());
       return;
   }
   // ...
}
```

Half-serialization

Sometimes having the queue-drain, `SerializedSubscriber` or `SerializedObserver` is a bit of an overkill. Such cases include when there is only one thread calling `onNext` but other threads may call `onError` or `onComplete` concurrently. Example operators include `takeUntil` where the other source may "interrupt" the main sequence and inject an `onComplete` into it before the main source itself would complete some time later. This is what I call **half-serialization**.

The approach uses the concepts of the deferred actions and atomic error management discussed above and has 3 methods for the `onNext`, `onError` and `onComplete` management:

```java
public static <T> void onNext(Subscriber<? super T> subscriber, T value,
        AtomicInteger wip, AtomicThrowable error) {
    if (wip.get() == 0 && wip.compareAndSet(0, 1)) {
        subscriber.onNext(value);
        if (wip.decrementAndGet() != 0) {
            Throwable ex = error.terminate();
            if (ex != null) {
                subscriber.onError(ex);
            } else {
                subscriber.onComplete();
            }
        }
    }
}

public static void onError(Subscriber<?> subscriber, Throwable ex,
        AtomicInteger wip, AtomicThrowable error) {
    if (error.addThrowable(ex)) {
        if (wip.getAndIncrement() == 0) {
            subscriber.onError(error.terminate());
        }
    } else {
        RxJavaPlugins.onError(ex);
    }
}

public static void onComplete(Subscriber<?> subscriber,
        AtomicInteger wip, AtomicThrowable error) {
    if (wip.getAndIncrement() == 0) {
        Throwable ex = error.terminate();
        if (ex != null) {
            subscriber.onError(ex);
        } else {
            subscriber.onComplete();
        }
    }
}
```

Here, the `wip` counter indicates there is an active emission happening and if found non-zero when trying to leave the `onNext`, it is taken as indication there was a concurrent `onError` or `onComplete()` call and the child must be notified. All subsequent calls to any of these methods are ignored. In this case, the `wip` is never decremented back to zero.

RxJava 2.x, again, supports these with the (internal) utility class `HalfSerializer` and allows targeting `Subscribers` and `Observers` with it.

Fast-path queue-drain

In some operators, it is unlikely concurrent threads try to enter into the drain loop at the same time and having to play the full enqueue-increment-dequeue adds unnecessary overhead.

Luckily, such situations can be detected by a simple compare-and-set attempt on the work-in-progress counter, trying to change the amount from 0 to 1. If it fails, there is a concurrent drain in progress and we revert back to the classical queue-drain logic. If succeeds, we don't enqueue anything but emit the value / perform the action right there and we try to leave the serialized section.

```java
public void onNext(T v) {
   if (wip.get() == 0 && wip.compareAndSet(0, 1)) {
       child.onNext(v);
       if (wip.decrementAndGet() == 0) {
           break;
       }
   } else {
       queue.offer(v);
       if (wip.getAndIncrement() != 0) {
           break;
       }
   }
   drainLoop();
}

void drain() {
   if (getAndIncrement() == 0) {
       drainLoop();
   }
}

void drainLoop() {
   // the usual drain loop part after the classical getAndIncrement()
}
```

In this pattern, the classical `drain` is spit into `drain` and `drainLoop`. The new `drain` does the increment-check and calls `drainLoop` and `drainLoop` contains the remaining logic with the loop, emission and wip management as usual.

On the fast path, when we try to leave it, it is possible a concurrent call to `onNext` or `drain` incremented the `wip` counter further and the decrement didn't return it to zero. This is an indication for further work and we call `drainLoop` to process it.

FlowableSubscriber

Version 2.0.7 introduced a new interface, `FlowableSubscriber` that extends `Subscriber` from Reactive-Streams. It has the same methods with the same parameter types but different textual rules attached to it, a set of relaxations to the Reactive-Streams specification to enable better performing RxJava internals while still honoring the specification to the letter for non-RxJava consumers of `Flowable`s.

The rule relaxations are as follows:

- §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`.

When a `Flowable` gets subscribed by a `Subscriber`, an `instanceof` check will detect `FlowableSubscriber` and not apply the `StrictSubscriber` wrapper that makes sure the relaxations don't happen. In practice, ensuring rule §3.9 has the most overhead because a bad request may happen concurrently with an emission of any normal event and thus has to be serialized with one of the methods described in previous sections.

In fact, 2.x was always implemented in this relaxed manner thus looking at existing code and style is the way to go.

Therefore, it is strongly recommended one implements custom intermediate and end operators via `FlowableSubscriber`.

From a source operator's perspective, extending the `Flowable` class and implementing `subscribeActual` has no need for dispatching over the type of the `Subscriber`; the backing infrastructure already applies wrapping if necessary thus one can be sure in `subscribeActual(Subscriber<? super T> s)` the parameter `s` is a `FlowableSubscriber`. (The signature couldn't be changed for compatibility reasons.) Since the two interfaces on the Java level are the same, no real preferential treating is necessary within sources (i.e., don't cast `s` into `FlowableSubscriber`.

The other base reactive consumers, `Observer`, `SingleObserver`, `MaybeObserver` and `CompletableObserver` don't need such relaxation, because

- they are only defined and used in RxJava (i.e., no other library implemented with them),

- they were conceptionally always derived from the relaxed `Subscriber` RxJava had,
- they don't have backpressure thus no `request()` call that would introduce another concurrency to think about,
- there is no way to trigger emission from upstream before `onSubscribe(Disposable)` returns in standard operators (again, no `request()` method).

Backpressure and cancellation

Backpressure (or flow control) in Reactive-Streams is the means to tell the upstream how many elements to produce or to tell it to stop producing elements altogether. Unlike the name suggest, there is no physical pressure preventing the upstream from calling `onNext` but the protocol to honor the request amount.

Replenishing

When dealing with basic transformations in a flow, there are often cases when the number of items the upstream sends should be different what the downstream receives. Some operators may want to filter out certain items, others would batch up items before sending one item to the downstream.

However, when an item is not forwarded by an operator, the downstream has no way of knowing its `request(1)` triggered an item generation that got dropped/buffered. Therefore, it can't know to (nor should it) repeat `request(1)` to "nudge" the source somewhere more upstream to try producing another item which now hopefully will result in an item being received by the downstream. Unlike, say the ACK-NACK based protocols, the requesting specified by the Reactive Streams are to be treated as cumulative. In the previous example, an impatient downstream would have 2 outstanding requests.

Therefore, if an operator is not guaranteed to relay an upstream item to downstream, and thus keeping a 1:1 ratio, it has the duty to keep requesting items from the upstream until said operator ends up in a position to supply an item to the downstream.

This may sound a bit complicated, but perhaps a demonstration of a `filter` operator can help:

```
final class FilterOddSubscriber implements FlowableSubscriber<Integer>, Subscription {

    final Subscriber<? super Integer> downstream;

    Subscription upstream;

    // ...

    @Override
    public void onSubscribe(Subscription s) {
        if (upstream != null) {
            s.cancel();
        } else {
            upstream = s;
            downstream.onSubscribe(this);
        }
    }

    @Override
    public void onNext(Integer item) {
        if (item % 2 != 0) {
           downstream.onNext(item);
        } else {
           upstream.request(1);
        }
    }

    @Override
    public void request(long n) {
        upstream.request(n);
    }

    // the rest omitted for brevity
}
```

In such operators, thee downstream's `request` calls are forwarded to the upstream as is, and for `n` times (at most, unless completed) the `onNext` will be invoked. In this operator, we look for the odd numbers of the flow. If we find one, the downstream will be notified. If the incoming item is even, we won't forward it to the downstream. However, the downstream is still expecting at least 1 item, but since the upstream and downstream practically talk to each other directly, the upstream considers its duty to generate items fulfilled. This misalignment is then resolved by requesting 1 more item from the upstream for the previous ignored item. If more items arrive that get ignored, more will be requested as replenishment.

Given that backpressure involves some overhead in the form of one or more atomic operations, requesting one by one could add a lot of overhead if the number of items filtered out is significantly more than those that passed through. If necessary, this situation can be either solved by decoupling the upstream and downstream's request management or using an RxJava-specific type and protocol extension in the form of ConditionalSubscribers.

Stable prefetching

In a previous section, we saw primitives to deal with request accounting and delayed `Subscriptions`, but often, operators have to react to request amount changes as well. This comes up when the operator has to decouple the downstream request amount from the amount it requests from upstream, such as `observeOn`.

Such logic can get quite complicated in operators but one of the simplest manifestation can be the `rebatchRequest` operator that combines request management with serialization to ensure that upstream is requested with a predictable pattern no matter how the downstream requested (less, more or even unbounded):

```
final class RebatchRequests<T> extends AtomicInteger
implements FlowableSubscriber<T>, Subscription {

    final Subscriber<? super T> child;

    final AtomicLong requested;

    final SpscArrayQueue<T> queue;

    final int batchSize;

    final int limit;

    Subscription s;

    volatile boolean done;
    Throwable error;

    volatile boolean cancelled;

    long emitted;

    public RebatchRequests(Subscriber<? super T> child, int batchSize) {
        this.child = child;
        this.batchSize = batchSize;
        this.limit = batchSize - (batchSize >> 2); // 75% of batchSize
        this.requested = new AtomicLong();
        this.queue = new SpscArrayQueue<T>(batchSize);
    }

    @Override
    public void onSubscribe(Subscription s) {
        this.s = s;
        child.onSubscribe(this);
        s.request(batchSize);
    }

    @Override
    public void onNext(T t) {
        queue.offer(t);
        drain();
    }

    @Override
    public void onError(Throwable t) {
        error = t;
        done = true;
        drain();
    }

    @Override
    public void onComplete() {
        done = true;
        drain();
    }

    @Override
    public void request(long n) {
        BackpressureHelper.add(requested, n);
        drain();
    }

    @Override
    public void cancel() {
        cancelled = true;
        s.cancel();
    }

    void drain() {
        // see next code example
    }
}
```

Here we extend `AtomicInteger` since the work-in-progress counting happens more often and is worth avoiding the extra indirection. The class extends `Subscription` and it hands itself to the `child Subscriber` to capture its `request()` (and `cancel()`) calls and route it to the main `drain` logic. Some operators need only this, some other operators (such as `observeOn` not only routes the downstream request but also does extra cancellations (cancels the asynchrony providing `Worker` as well) in its `cancel()` method.

Important: when implementing operators for `Flowable` and `Observable` in

RxJava 2.x, you are not allowed to pass along an upstream `Subscription` or `Disposable` to the child `Subscriber`/`Observer` when the operator logic itself doesn't require hooking the `request`/`cancel`/`dispose` calls. The reason for this is how operator-fusion is implemented on top of `Subscription` and `Disposable` passing through `onSubscribe` in RxJava 2.x (and in **Reactor 3**). See the next section about operator-fusion. There is no fusion in `Single`, `Completable` or `Maybe` (because there is no requesting or unbounded buffering with them) and their operators can pass the upstream `Disposable` along as is.

Next comes the `drain` method whose pattern appears in many operators (with slight variations on how and what the emission does).

```
void drain() {
    if (getAndIncrement() != 0) {
        return;
    }

    int missed = 1;

    for (;;) {
        long r = requested.get();
        long e = 0L;
        long f = emitted;

        while (e != r) {
            if (cancelled) {
                return;
            }
            boolean d = done;

            if (d) {
                Throwable ex = error;
                if (ex != null) {
                    child.onError(ex);
                    return;
                }
            }

            T v = queue.poll();
            boolean empty = v == null;

            if (d && empty) {
                child.onComplete();
                return;
            }

            if (empty) {
                break;
            }

            child.onNext(v);

            e++;
            if (++f == limit) {
               s.request(f);
               f = 0L;
            }
        }

        if (e == r) {
            if (cancelled) {
                return;
            }

            if (done) {
                Throwable ex = error;
                if (ex != null) {
                    child.onError(ex);
                    return;
                }
                if (queue.isEmpty()) {
                    child.onComplete();
                    return;
                }
            }
        }

        if (e != 0L) {
            BackpressureHelper.produced(requested, e);
        }

        emitted = f;
        missed = addAndGet(-missed);
        if (missed == 0) {
            break;
        }
    }
}
```

This particular pattern is called the **stable-request queue-drain**. Another variation doesn't care about request amount stability towards upstream and simply requests the amount it delivered to the child:

```
void drain() {
```

```java
if (getAndIncrement() != 0) {
    return;
}

int missed = 1;

for (;;) {
    long r = requested.get();
    long e = 0L;

    while (e != r) {
        if (cancelled) {
            return;
        }
        boolean d = done;

        if (d) {
            Throwable ex = error;
            if (ex != null) {
                child.onError(ex);
                return;
            }
        }

        T v = queue.poll();
        boolean empty = v == null;

        if (d && empty) {
            child.onComplete();
            return;
        }

        if (empty) {
            break;
        }

        child.onNext(v);

        e++;
    }

    if (e == r) {
        if (cancelled) {
            return;
        }
```

```
            if (done) {
                Throwable ex = error;
                if (ex != null) {
                    child.onError(ex);
                    return;
                }
                if (queue.isEmpty()) {
                    child.onComplete();
                    return;
                }
            }
        }

        if (e != 0L) {
            BackpressureHelper.produced(requested, e);
            s.request(e);
        }

        missed = addAndGet(-missed);
        if (missed == 0) {
            break;
        }
    }
}
```

The third variation allows delaying a potential error until the upstream has terminated and all normal elements have been delivered to the child:

```
final boolean delayError;

void drain() {
    if (getAndIncrement() != 0) {
        return;
    }

    int missed = 1;

    for (;;) {
        long r = requested.get();
        long e = 0L;

        while (e != r) {
            if (cancelled) {
                return;
            }
            boolean d = done;
            if (d && !delayError) {
                Throwable ex = error;
                if (ex != null) {
                    child.onError(ex);
                    return;
                }
            }

            T v = queue.poll();
            boolean empty = v == null;

            if (d && empty) {
                Throwable ex = error;
                if (ex != null) {
                    child.onError(ex);
                } else {
                    child.onComplete();
                }
                return;
            }

            if (empty) {
                break;
            }

            child.onNext(v);

            e++;
        }

        if (e == r) {
            if (cancelled) {
                return;
            }

            if (done) {
                if (delayError) {
                    if (queue.isEmpty()) {
                        Throwable ex = error;
                        if (ex != null) {
                            child.onError(ex);
                        } else {
                            child.onComplete();
                        }
                        return;
                    }
```

```
                } else {
                    Throwable ex = error;
                    if (ex != null) {
                        child.onError(ex);
                        return;
                    }
                    if (queue.isEmpty()) {
                        child.onComplete();
                        return;
                    }
                }
            }
        }

        if (e != 0L) {
            BackpressureHelper.produced(requested, e);
            s.request(e);
        }

        missed = addAndGet(-missed);
        if (missed == 0) {
            break;
        }
    }
}
```

If the downstream cancels the operator, the `queue` may still hold elements which may get referenced longer than expected if the operator chain itself is referenced in some way. On the user level, applying `onTerminateDetach` will forget all references going upstream and downstream and can help with this situation. On the operator level, RxJava 2.x usually calls `clear()` on the `queue` when the sequence is cancelled or ends before the queue is drained naturally. This requires some slight change to the drain loop:

```
final boolean delayError;

@Override
public void cancel() {
    cancelled = true;
    s.cancel();
    if (getAndIncrement() == 0) {
        queue.clear();    // <-----------------------------
    }
}

void drain() {
    if (getAndIncrement() != 0) {
        return;
    }

    int missed = 1;

    for (;;) {
        long r = requested.get();
        long e = 0L;

        while (e != r) {
            if (cancelled) {
                queue.clear();     // <-----------------------------
                return;
            }
            boolean d = done;

            if (d && !delayError) {
                Throwable ex = error;
                if (ex != null) {
                    queue.clear();     // <-----------------------------
                    child.onError(ex);
                    return;
                }
            }

            T v = queue.poll();
            boolean empty = v == null;

            if (d && empty) {
                Throwable ex = error;
                if (ex != null) {
                    child.onError(ex);
                } else {
                    child.onComplete();
                }
                return;
            }

            if (empty) {
                break;
            }

            child.onNext(v);

            e++;
        }
        if (e == r) {
            if (cancelled) {
                queue.clear();     // <-----------------------------
                return;
            }

            if (done) {
                if (delayError) {
                    if (queue.isEmpty()) {
                        Throwable ex = error;
                        if (ex != null) {
                            child.onError(ex);
                        } else {
                            child.onComplete();
                        }
                        return;
                    }
                } else {
                    Throwable ex = error;
                    if (ex != null) {
                        queue.clear();     // <-----------------------------
                        child.onError(ex);
                        return;
                    }
                    if (queue.isEmpty()) {
                        child.onComplete();
                        return;
                    }
                }
            }
        }

        if (e != 0L) {
            BackpressureHelper.produced(requested, e);
            s.request(e);
        }

        missed = addAndGet(-missed);
        if (missed == 0) {
            break;
        }
    }
}
```

Since the queue is single-producer-single-consumer, its `clear()` must be called from a single thread - which is provided by the serialization loop and is enabled by the `getAndIncrement() == 0` "half-loop" inside `cancel()`.

An important note on the order of calls to `done` and the `queue`'s state:

```
boolean d = done;
T v = queue.poll();
```

and

```
boolean d = done;
boolean empty = queue.isEmpty();
```

These must happen in the order specified. If they were swapped, it is possible when the drain runs asynchronously to an `onNext`/`onComplete()`, the queue may appear empty at first, then it gets elements followed by `done = true` and a late `done` check in the drain loop may complete the sequence thinking it delivered all values there was.

Single valued results

Sometimes an operator only emits one single value at some point instead of emitting more or all of its sources. Such operators include `fromCallable`, `reduce`, `any`, `all`, `first`, etc.

The classical queue-drain works here but is a bit of an overkill to allocate objects to store the work-in-progress counter, request accounting and the queue itself. These elements can be reduced to a single state-machine with one state counter object - often inlinded by extending AtomicInteger - and a plain field for storing the single value to be emitted.

The state machine handing the possible concurrent downstream requests and normal completion path is a bit complicated to show here and is quite easy to get wrong.

RxJava 2.x supports this kind of behavior through the (internal) `DeferredScalarSubscription` for operators without an upstream source (`fromCallable`) and the (internal) `DeferredScalarSubscriber` for reduce-like operators with an upstream source.

Using the `DeferredScalarSubscription` is straightforward, one creates it, sends it to the downstream via `onSubscribe` and later on calls `complete(T)` to signal the end with a single value:

```
DeferredScalarSubscription<Integer> dss = new DeferredScalarSubscription<>(child);
child.onSubscribe(dss);

dss.complete(1);
```

Using the `DeferredScalarSubscriber` requires more coding and extending the class itself:

```java
final class Counter extends DeferredScalarSubscriber<Object, Integer> {
   public Counter(Subscriber<? super Integer> child) {
       super(child);
       value = 0;
       hasValue = true;
   }

   @Override
   public void onNext(Object t) {
       value++;
   }
}
```

By default, the `DeferredScalarSubscriber.onSubscribe()` requests `Long.MAX_VALUE` from the upstream (but the method can be overridden in subclasses).

Single-element post-complete

Some operators have to modulate a sequence of elements in a 1:1 fashion but when the upstream terminates, they need to produce a final element followed by a terminal event (usually `onComplete`).

```java
final class OnCompleteEndWith implements Subscriber<T>, Subscription {
    final Subscriber<? super T> child;

    final T finalElement;

    Subscription s;

    public OnCompleteEndWith(Subscriber<? super T> child, T finalElement) {
        this.child = child;
        this.finalElement = finalElement;
    }

    @Override
    public void onSubscribe(Subscription s) {
        this.s = s;
        child.onSubscribe(this);
    }

    @Override
    public void onNext(T t) {
        child.onNext(t);
    }

    @Overide
    public void onError(Throwable t) {
        child.onError(t);
    }

    @Override
    public void onComplete() {
        child.onNext(finalElement);
        child.onComplete();
    }

    @Override
    public void request(long n) {
        s.request(n);
    }

    @Override
    public void cancel() {
        s.cancel();
    }
}
```

This works if the downstream request more than the upstream produces + 1, otherwise the call to `onComplete` may overflow the child `Subscriber`.

Heavyweight solutions such as queue-drain or `SubscriptionArbiter` with `ScalarSubscriber` can be used here, however, there is a more elegant solution to the problem.

The idea is that request amounts occupy only 63 bits of a 64 bit (atomic) long type. If we'd mask out the lower 63 bits when working with the amount, we can use the most significant bit to indicate the upstream sequence has finished and then on, any 0 to n request amount change can trigger the emission of the `finalElement`. Since a downstream `request()` can race with an upstream `onComplete`, marking the bit atomically via a compare-and-set ensures correct state transition.

For this, the `OnCompleteEndWith` has to be changed by adding an `AtomicLong` for accounting requests, a long for counting the production, then updating `request()` and `onComplete()` methods:

```
final class OnCompleteEndWith
extends AtomicLong
implements FlowableSubscriber<T>, Subscription {
    final Subscriber<? super T> child;

    final T finalElement;
```

```
Subscription s;

long produced;

static final class long REQUEST_MASK = Long.MAX_VALUE;  // 0b01111...111L
static final class long COMPLETE_MASK = Long.MIN_VALUE; // 0b10000...000L

public OnCompleteEndWith(Subscriber<? super T> child, T finalElement) {
    this.child = child;
    this.finalElement = finalElement;
}

@Override
public void onSubscribe(Subscription s) { ... }

@Override
public void onNext(T t) {
    produced++;                 // <-------------------------
    child.onNext(t);
}

@Overide
public void onError(Throwable t) { ... }

@Override
public void onComplete() {
    long p = produced;
    if (p != 0L) {
        produced = 0L;
        BackpressureHelper.produced(this, p);
    }

    for (;;) {
        long current = get();
        if ((current & COMPLETE_MASK) != 0) {
            break;
        }
        if ((current & REQUEST_MASK) != 0) {
            lazySet(Long.MIN_VALUE + 1);
            child.onNext(finalElement);
            child.onComplete();
            return;
        }
        if (compareAndSet(current, COMPLETE_MASK)) {
            break;
        }
        }
    }

    @Override
    public void request(long n) {
        for (;;) {
            long current = get();
            if ((current & COMPLETE_MASK) != 0) {
                if (compareAndSet(current, COMPLETE_MASK + 1)) {
                    child.onNext(finalElement);
                    child.onComplete();
                }
                break;
            }
            long u = BackpressureHelper.addCap(current, n);
            if (compareAndSet(current, u)) {
                s.request(n);
                break;
            }
        }
    }

    @Override
    public void cancel() { ... }
}
```

RxJava 2 has a couple of operators, `materialize`, `mapNotification`, `onErrorReturn`, that require this type of behavior and for that, the (internal) `SinglePostCompleteSubscriber` class captures the algorithms above:

```
final class OnCompleteEndWith<T> extends SinglePostCompleteSubscriber<T, T> {
    final Subscriber<? super T> child;

    public OnCompleteEndWith(Subscriber<? super T> child, T finalElement) {
        this.child = child;
        this.value = finalElement;
    }

    @Override
    public void onNext(T t) {
        produced++;                 // <-------------------------
        child.onNext(t);
    }

    @Overide
    public void onError(Throwable t) { ... }
    @Override
    public void onComplete() {
        complete(value);
    }
}
```

Multi-element post-complete

Certain operators may need to emit multiple elements after the main sequence completes, which may or may not relay elements from the live upstream before its termination. An example operator is `buffer(int, int)` when the skip < size yielding overlapping buffers. In this operator, it is possible when the upstream completes, several overlapping buffers are waiting to be emitted to the child but that has to happen only when the child actually requested more buffers.

The state machine for this case is complicated but RxJava has two (internal) utility methods on `QueueDrainHelper` for dealing with the situation:

```
<T> void postComplete(Subscriber<? super T> actual,
                      Queue<T> queue,
                      AtomicLong state,
                      BooleanSupplier isCancelled);

<T> boolean postCompleteRequest(long n,
                                Subscriber<? super T> actual,
                                Queue<T> queue,
                                AtomicLong state,
                                BooleanSupplier isCancelled);
```

They take the child `Subscriber`, the queue to drain from, the state holding the current request amount and a callback to see if the downstream cancelled the sequence.

Usage of these methods is as follows:

```
final class EmitTwice<T> extends AtomicLong
implements FlowableSubscriber<T>, Subscription, BooleanSupplier {
    final Subscriber<? super T> child;

    final ArrayDeque<T> buffer;

    volatile boolean cancelled;

    Subscription s;

    long produced;

    public EmitTwice(Subscriber<? super T> child) {
```

```java
        this.child = child;
        this.buffer = new ArrayDeque<>();
    }

    @Override
    public void onSubscribe(Subscription s) {
        this.s = s;
        child.onSubscribe(this);
    }

    @Override
    public void onNext(T t) {
        produced++;
        buffer.offer(t);
        child.onNext(t);
    }

    @Override
    public void onError(Throwable t) {
        buffer.clear();
        child.onError(t);
    }

    @Override
    public void onComplete() {
        long p = produced;
        if (p != 0L) {
            produced = 0L;
            BackpressureHelper.produced(this, p);
        }
        QueueDrainHelper.postComplete(child, buffer, this, this);
    }

    @Override
    public boolean getAsBoolean() {
        return cancelled;
    }

    @Override
    public void cancel() {
        cancelled = true;
        s.cancel();
    }

    @Override
    public void request(long n) {
        if (!QueueDrainHelper.postCompleteRequest(n, child, buffer, this, this)) {
            s.request(n);
        }
    }
}
```

Creating operator classes

Creating operator implementations in 2.x is simpler than in 1.x and incurs less allocation as well. You have the choice to implement your operator as a `Subscriber`-transformer to be used via `lift` or as a fully-fledged base reactive class.

Operator by extending a base reactive class

In 1.x, extending `Observable` was possible but convoluted because you had to implement the `OnSubscribe` interface separately and pass it to `Observable.create()` or to the `Observable(OnSubscribe)` protected constructor.

In 2.x, all base reactive classes are abstract and you can extend them directly without any additional indirection:

```
public final class FlowableMyOperator extends Flowable<Integer> {
    final Publisher<Integer> source;

    public FlowableMyOperator(Publisher<Integer> source) {
        this.source = source;
    }

    @Override
    protected void subscribeActual(Subscriber<? super Integer> s) {
        source.map(v -> v + 1).subscribe(s);
    }
}
```

When taking other reactive types as inputs in these operators, it is recommended one defines the base reactive interfaces instead of the abstract classes, allowing better interoperability between libraries (especially with `Flowable` operators and other Reactive-Streams `Publisher`s). To recap, these are the class-interface pairs:

- `Flowable` - `Publisher` - `FlowableSubscriber`/`Subscriber`
- `Observable` - `ObservableSource` - `Observer`
- `Single` - `SingleSource` - `SingleObserver`
- `Completable` - `CompletableSource` - `CompletableObserver`
- `Maybe` - `MaybeSource` - `MaybeObserver`

RxJava 2.x locks down `Flowable.subscribe` (and the same methods in the other types) in order to provide runtime hooks into the various flows, therefore, implementors are given the `subscribeActual()` to be overridden. When it is invoked, all relevant hooks and wrappers have been applied. Implementors should avoid throwing unchecked exceptions as the library generally can't deliver it to the respective `Subscriber` due to lifecycle restrictions of the Reactive-Streams specification and sends it to the global error consumer via `RxJavaPlugins.onError`.

Unlike in 1.x, In the example above, the incoming `Subscriber` is simply used directly for subscribing again (but still at most once) without any kind of wrapping. In 1.x, one needs to call `Subscribers.wrap` to avoid double calls to `onStart` and cause unexpected double initialization or double-requesting.

Unless one contributes a new operator to RxJava, working with such classes may become tedious, especially if they are intermediate operators:

```
new FlowableThenSome(
    new FlowableOther(
        new FlowableMyOperator(Flowable.range(1, 10).map(v -> v * v))
    )
)
```

This is an unfortunate effect of Java lacking extension method support. A possible ease on this burden is by using `compose` to have fluent inline application of the custom operator:

```
Flowable.range(1, 10).map(v -> v * v)
.compose(f -> new FlowableOperatorWithParameter(f, 10));

Flowable.range(1, 10).map(v -> v * v)
.compose(FlowableMyOperator::new);
```

Operator targeting lift()

The alternative to the fluent application problem is to have a `Subscription`-transformer implemented instead of extending the whole reactive base class and use the respective type's `lift()` operator to get it into the sequence.

First one has to implement the respective `XOperator` interface:

```
public final class MyOperator implements FlowableOperator<Integer, Integer> {

    @Override
    public Subscriber<? super Integer> apply(Subscriber<? super Integer> child) {
        return new Op(child);
    }

    static final class Op implements FlowableSubscriber<Integer>, Subscription {
        final Subscriber<? super Integer> child;
        Subscription s;

        public Op(Subscriber<? super Integer> child) {
            this.child = child;
        }

        @Override
        pubic void onSubscribe(Subscription s) {
            this.s = s;
            child.onSubscribe(this);
        }

        @Override
        public void onNext(Integer v) {
            child.onNext(v * v);
        }

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

        @Override
        public void onComplete() {
            child.onComplete();
        }

        @Override
        public void cancel() {
            s.cancel();
        }

        @Override
        public void request(long n) {
            s.request(n);
        }
    }
}
```

You may recognize that implementing operators via extension or lifting looks quite similar. In both cases, one usually implements a `FlowableSubscriber` (`Observer`, etc) that takes a downstream `Subscriber`, implements the business logic in the `onXXX` methods and somehow (manually or as part of `lift()`'s lifecycle) gets subscribed to an upstream source.

The benefit of applying the Reactive-Streams design to all base reactive types is that each consumer type is now an interface and can be applied to operators that have to extend some class. This was a pain in 1.x because `Subscriber` and `SingleSubscriber` are classes themselves, plus `Subscriber.request()` is a protected-final method and an operator's `Subscriber` can't implement the `Producer` interface at the same time. In 2.x there is no such problem and one can have both `Subscriber`, `Subscription` or even `Observer` together in the same consumer type.

Operator fusion

Operator fusion has the premise that certain operators can be combined into one single operator (macro-fusion) or their internal data structures shared between each other (micro-fusion) that allows fewer allocations, lower overhead and better performance.

This advanced concept was invented, worked out and studied in the Reactive-Streams-Commons research project manned by the leads of RxJava and Project Reactor. Both libraries use the results in their implementation, which look the same but are incompatible due to different classes and packages involved. In addition, RxJava 2.x's approach is a more polished version of the invention due to delays between the two project's development.

Since operator-fusion is optional, you may chose to not bother making your operator fusion-enabled. The `DeferredScalarSubscription` is fusion-enabled and needs no additional development in this regard though.

If you chose to ignore operator-fusion, you still have to follow the requirement of never forwarding a `Subscription/Disposable` coming through `onSubscribe` of `Subscriber/Observer` as this may break the fusion protocol and may skip your operator's business logic entirely:

```java
final class SomeOp<T> implements Subscriber<T>, Subscription {

    // ...
    Subscription s;

    public void onSubscribe(Subscription s) {
        this.s = s;
        child.onSubscribe(this);                // <---------------------------
    }

    @Override
    public void cancel() {
        s.cancel();
    }

    @Override
    public void request(long n) {
        s.request(n);
    }

    // ...
}
```

Yes, this adds one more indirection between operators but it is still cheap (and would be necessary for the operator anyway) but enables huge performance gains with the right chain of operators.

Generations

Given this novel approach, a generation number can be assigned to various implementation styles of reactive architectures:

Generation 0 These are the classical libraries that either use `java.util.Observable` or are listener based (Java Swing's `ActionListener`). Their common property is that they don't support composition (of events and cancellation). See also **Google Agera**.

Generation 1 This is the level of the **Rx.NET** library (even up to 3.x) that supports composition, but has no notion for backpressure and doesn't properly support synchronous cancellation. Many JavaScript libraries such as **RxJS 5** are still on this level. See also **Google gRPC**.

Generation 2 This is what **RxJava 1.x** is categorized, it supports composition, backpressure and synchronous cancellation along with the ability to lift an operator into a sequence.

Generation 3 This is the level of the Reactive-Streams based libraries such as **Reactor 2** and **Akka-Stream**. They are based upon a specification that evolved out of RxJava but left behind its drawbacks (such as the need to return anything from `subscribe()`). This is incompatible with RxJava 1.x and thus 2.x had to be rewritten from scratch.

Generation 4 This level expands upon the Reactive-Streams interfaces with operator-fusion (in a compatible fashion, that is, op-fusion is optional between two stages and works without them). **Reactor 3** and **RxJava 2** are at this level. The material around **Akka-Stream** mentions operator-fusion as well, however, **Akka-Stream** is not a native Reactive-Streams implementation (requires a materializer to get a `Publisher` out) and as such it is only Gen 3.

There are discussions among the 4th generation library providers to have the elements of operator-fusion standardized in Reactive-Streams 2.0 specification (or in a neighboring extension) and have **RxJava 3** and **Reactor 4** work together on that aspect as well.

Components

Callable and ScalarCallable

Certain `Flowable` sources, similar to `Single` or `Completable` are known to ever emit zero or one item and that single item is known to be constant or is computed synchronously. Well known examples of this are `just()`, `empty()` and `fromCallable`. Subscribing to these sources, like any other sources, adds the same infrastructure overhead which can often be avoided if the consumer could just pick or have the item calculated on the spot.

For example, `just` and `empty` appears as the mapping result of a `flatMap` operation:

```
source.flatMap(v -> {
    if (v % 2 == 0) {
        return just(v);
    }
    return empty();
})
```

Here, if we'd somehow recognize that `empty()` won't emit a value but only `onComplete` we could simply avoid subscribing to it inside `flatMap`, saving on the overhead. Similarly, recognizing that `just` emits exactly one item we can route it differently inside `flatMap` and again, avoiding creating a lot of objects to get to the same single item.

In other times, knowing the emission property can simplify or chose a different operator instead of the applied one. For example, applying `flatMap` to an `empty()` source has no use since there won't be any item to be flattened into a sequence; the whole flattened sequence is going to be empty. Knowing that a source is `just` to `flatMap`, there is no need for the complicated inner mechanisms as there is going to be only one mapped inner source and one can subscribe the downstream's `Subscriber` to it directly.

```
Flowable.just(1).flatMap(v -> Flowable.range(v, 5)).subscribe(...);

// in some specialized operator:

T value; // from just()

@Override
public void subscribeActual(Subscriber<? super T> s) {
    mapper.apply(value).subscribe(s);
}
```

There could be other sources with these properties, therefore, RxJava 2 uses the `io.reactivex.internal.fusion.ScalarCallable` and `java.util.Callable` interfaces to indicate a source is a constant or sequentially computable. When a source `Flowable` or `Observable` is marked with one of these interfaces, many fusion enabled operators will perform special actions to avoid the overhead of a normal and general source.

We use Java's own and preexisting `java.util.Callable` interface to indicate a synchronously computable source. The `ScalarCallable` is an extension to this interface by which it suppresses the `throws Exception` of `Callable.call()`:

```java
interface Callable<T> {
    T call() throws Exception;
}

interface ScalarCallable<T> extends Callable<T> {
    @Override
    T call();
}
```

The reason for the two separate interfaces is that if a source is constant, like `just`, one can perform assembly-time optimizations with it knowing that each regular `subscribe` invocation would have resulted in the same single value.

`Callable` denotes sources, such as `fromCallable` that indicates the single value has to be calculated at runtime of the flow. By this logic, you can see that `ScalarCallable` is a `Callable` on its own right because the constant can be "calculated" as late as the runtime phase of the flow.

Since Reactive-Streams forbids using `nulls` as emission values, we can use `null` in `(Scalar)Callable` marked sources to indicate there is no value to be emitted, thus one can't mistake an user's `null` with the empty indicator `null`. For example, this is how `empty()` is implemented:

```java
final class FlowableEmpty extends Flowable<Object> implements ScalarCallable<Object> {
    @Override
    public void subscribeActual(Subscriber<? super T> s) {
        EmptySubscription.complete(s);
    }

    @Override
    public Object call() {
        return null; // interpreted as no value available
    }
}
```

Sources implementing `Callable` may throw checked exceptions from `call()` which is handled by the consumer operators as an indication to signal `onError` in an operator specific manner (such as delayed).

```java
final class FlowableIOException extends Flowable<Object> implements Callable<Object> {
    @Override
    public void subscribeActual(Subscriber<? super T> s) {
        EmptySubscription.error(new IOException(), s);
    }

    @Override
    public Object call() throws Exception {
        throw new IOException();
    }
}
```

However, implementors of `ScalarCallable` should avoid throwing any exception and limit the code in `call()` be constant or simple computation that can be legally executed during assembly time.

As the consumer of sources, one may want to deal with such kind of special `Flowables` or `Observables`. For example, if you create an operator that can leverage the knowledge of a single element source as its main input, you can check the types and extract the value of a `ScalarCallable` at assembly time right in the operator:

```java
// Flowable.java
public final Flowable<Integer> plusOne() {
    if (this instanceof ScalarCallable) {
        Integer value = ((ScalarCallable<Integer>)this).call();
        if (value == null) {
            return empty();
        }
        return just(value + 1);
    }
    return cast(Integer.class).map(v -> v + 1);
}
```

or as a `FlowableTransformer`:

```java
FlowableTransformer<Integer, Integer> plusOneTransformer = source -> {
    if (source instanceof ScalarCallable) {
        Integer value = ((ScalarCallable<Integer>)source).call();
        if (value == null) {
            return empty();
        }
        return just(value + 1);
    }
    return source.map(v -> v + 1);
};
```

However, it is not mandatory to handle `ScalarCallables` and `Callables` separately. Since the former extends the latter, the type check can be deferred till subscription time and handled with the same code path:

```
final class FlowablePlusOne extends Flowable<Integer> {
    final Publisher<Integer> source;

    FlowablePlusOne(Publisher<Integer> source) {
        this.source = source;
    }

    @Override
    public void subscribeActual(Subscriber<? super Integer> s) {
        if (source instanceof Callable) {
            Integer value;

            try {
                value = ((Callable<Integer>)source).call();
            } catch (Throwable ex) {
                Exceptions.throwIfFatal(ex);
                EmptySubscription.error(ex, s);
                return;
            }

            s.onSubscribe(new ScalarSubscription<Integer>(s, value + 1));
        } else {
            new FlowableMap<>(source, v -> v + 1).subscribe(s);
        }
    }
}
```

ConditionalSubscriber

TBD

QueueSubscription and QueueDisposable

TBD

Example implementations

TBD

`map` + `filter` hybrid

TBD

Ordered `merge`

TBD