*Originally contributed to StackOverflow Documentation (going defunct) by @akarnokd, revised for version 2.x.*

# Introduction

**Backpressure** is when in an `Flowable` processing pipeline, some asynchronous stages can't process the values fast enough and need a way to tell the upstream producer to slow down.

The classic case of the need for backpressure is when the producer is a hot source:

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

source
.observeOn(Schedulers.computation())
.subscribe(v -> compute(v), Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}

Thread.sleep(10_000);
```

In this example, the main thread will produce 1 million items to an end consumer which is processing it on a background thread. It is likely the `compute(int)` method takes some time but the overhead of the `Flowable` operator chain may also add to the time it takes to process items. However, the producing thread with the for loop can't know this and keeps `onNext`ing.

Internally, asynchronous operators have buffers to hold such elements until they can be processed. In the classical Rx.NET and early RxJava, these buffers were unbounded, meaning that they would likely hold nearly all 1 million elements from the example. The problem starts when there are, for example, 1 billion elements or the same 1 million sequence appears 1000 times in a program, leading to `OutOfMemoryError` and generally slowdowns due to excessive GC overhead.

Similar to how error-handling became a first-class citizen and received operators to deal with it (via `onErrorXXX` operators), backpressure is another property of dataflows that the programmer has to think about and handle (via `onBackpressureXXX` operators).

Beyond the `PublishProcessor`above, there are other operators that don't support backpressure, mostly due to functional reasons. For example, the operator `interval` emits values periodically, backpressuring it would lead to shifting in the period relative to a wall clock.

In modern RxJava, most asynchronous operators now have a bounded internal buffer, like `observeOn` above and any attempt to overflow this buffer will terminate the whole sequence with `MissingBackpressureException`. The documentation of each operator has a description about its backpressure behavior.

However, backpressure is present more subtly in regular cold sequences (which don't and shouldn't yield `MissingBackpressureException`). If the first example is rewritten:

```
Flowable.range(1, 1_000_000)
.observeOn(Schedulers.computation())
.subscribe(v -> compute(v), Throwable::printStackTrace);

Thread.sleep(10_000);
```

There is no error and everything runs smoothly with small memory usage. The reason for this is that many source operators can "generate" values on demand and thus the operator `observeOn` can tell the `range` generate at most so many values the `observeOn` buffer can hold at once without overflow.

This negotiation is based on the computer science concept of co-routines (I call you, you call me). The operator `range` sends a callback, in the form of an implementation of the `org.reactivestreams.Subscription` interface, to the `observeOn` by calling its (inner `Subscriber`'s) `onSubscribe`. In return, the `observeOn` calls `Subscription.request(n)` with a value to tell the `range` it is allowed to produce (i.e., `onNext` it) that many **additional** elements. It is then the `observeOn`'s responsibility to call the `request` method in the right time and with the right value to keep the data flowing but not overflowing.

Expressing backpressure in end-consumers is rarely necessary (because they are synchronous in respect to their immediate upstream and backpressure naturally happens due to call-stack blocking), but it may be easier to understand the workings of it:

```
Flowable.range(1, 1_000_000)
.subscribe(new DisposableSubscriber<Integer>() {
    @Override
    public void onStart() {
        request(1);
    }

    public void onNext(Integer v) {
        compute(v);

        request(1);
    }

    @Override
    public void onError(Throwable ex) {
        ex.printStackTrace();
    }
    @Override
    public void onComplete() {
        System.out.println("Done!");
    }
});
```

Here the `onStart` implementation indicates `range` to produce its first value, which is then received in `onNext`. Once the `compute(int)` finishes, the another value is then requested from `range`. In a naive implementation of `range`, such call would recursively call `onNext`, leading to `StackOverflowError` which is of course undesirable.

To prevent this, operators use so-called trampolining logic that prevents such reentrant calls. In `range`'s terms, it will remember that there was a `request(1)` call while it called `onNext()` and once `onNext()` returns, it will make another round and call `onNext()` with the next integer value. Therefore, if the two are swapped, the example still works the same:

```
@Override
public void onNext(Integer v) {
    request(1);

    compute(v);
}
```

However, this is not true for `onStart`. Although the `Flowable` infrastructure guarantees it will be called at most once on each `Subscriber`, the call to `request(1)` may trigger the emission of an element right away. If one has initialization logic after the call to `request(1)` which is needed by `onNext`, you may end up with exceptions:

```
Flowable.range(1, 1_000_000)
.subscribe(new DisposableSubscriber<Integer>() {

    String name;

    @Override
    public void onStart() {
        request(1);

        name = "RangeExample";
    }

    @Override
    public void onNext(Integer v) {
        compute(name.length + v);
        request(1);
    }

    // ... rest is the same
});
```

In this synchronous case, a `NullPointerException` will be thrown immediately while still executing `onStart`. A more subtle bug happens if the call to `request(1)` triggers an asynchronous call to `onNext` on some other thread and reading `name` in `onNext` races writing it in `onStart` post `request`.

Therefore, one should do all field initialization in `onStart` or even before that and call `request()` last. Implementations of `request()` in operators ensure proper happens-before relation (or in other terms, memory release or full fence) when necessary.

# The onBackpressureXXX operators

Most developers encounter backpressure when their application fails with `MissingBackpressureException` and the exception usually points to the `observeOn` operator. The actual cause is usually the non-backpressured use of `PublishProcessor`, `timer()` or `interval()` or custom operators created via `create()`.

There are several ways of dealing with such situations.

## Increasing the buffer sizes

Sometimes such overflows happen due to bursty sources. Suddenly, the user taps the screen too quickly and `observeOn`'s default 16-element internal buffer on Android overflows.

Most backpressure-sensitive operators in the recent versions of RxJava now allow programmers to specify the size of their internal buffers. The relevant parameters are usually called `bufferSize`, `prefetch` or `capacityHint`. Given the overflowing example in the introduction, we can just increase the buffer size of `observeOn` to have enough room for all values.

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

source.observeOn(Schedulers.computation(), 1024 * 1024)
      .subscribe(e -> { }, Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

Note however that generally, this may be only a temporary fix as the overflow can still happen if the source overproduces the predicted buffer size. In this case, one can use one of the following operators.

## Batching/skipping values with standard operators

In case the source data can be processed more efficiently in batch, one can reduce the likelihood of MissingBackpressureException by using one of the standard batching operators (by size and/or by time).

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

source
      .buffer(1024)
      .observeOn(Schedulers.computation(), 1024)
      .subscribe(list -> {
          list.parallelStream().map(e -> e * e).first();
      }, Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

If some of the values can be safely ignored, one can use the sampling (with time or another Flowable) and throttling operators (throttleFirst, throttleLast, throttleWithTimeout).

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

source
      .sample(1, TimeUnit.MILLISECONDS)
      .observeOn(Schedulers.computation(), 1024)
      .subscribe(v -> compute(v), Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

Note hovewer that these operators only reduce the rate of value reception by the downstream and thus they may still lead to MissingBackpressureException.

## onBackpressureBuffer()

This operator in its parameterless form reintroduces an unbounded buffer between the upstream source and the downstream operator. Being unbounded means as long as the JVM doesn't run out of memory, it can handle almost any amount coming from a bursty source.

```
Flowable.range(1, 1_000_000)
          .onBackpressureBuffer()
          .observeOn(Schedulers.computation(), 8)
          .subscribe(e -> { }, Throwable::printStackTrace);
```

In this example, the `observeOn` goes with a very low buffer size yet there is no `MissingBackpressureException` as `onBackpressureBuffer` soaks up all the 1 million values and hands over small batches of it to `observeOn`.

Note however that `onBackpressureBuffer` consumes its source in an unbounded manner, that is, without applying any backpressure to it. This has the consequence that even a backpressure-supporting source such as `range` will be completely realized.

There are 4 additional overloads of `onBackpressureBuffer`

### onBackpressureBuffer(int capacity)

This is a bounded version that signals `BufferOverflowError`in case its buffer reaches the given capacity.

```
Flowable.range(1, 1_000_000)
          .onBackpressureBuffer(16)
          .observeOn(Schedulers.computation())
          .subscribe(e -> { }, Throwable::printStackTrace);
```

The relevance of this operator is decreasing as more and more operators now allow setting their buffer sizes. For the rest, this gives an opportunity to "extend their internal buffer" by having a larger number with `onBackpressureBuffer` than their default.

### onBackpressureBuffer(int capacity, Action onOverflow)

This overload calls a (shared) action in case an overflow happens. Its usefulness is rather limited as there is no other information provided about the overflow than the current call stack.

### onBackpressureBuffer(int capacity, Action onOverflow, BackpressureOverflowStrategy strategy)

This overload is actually more useful as it let's one define what to do in case the capacity has been reached. The `BackpressureOverflow.Strategy` is an interface actually but the class `BackpressureOverflow` offers 4 static fields with implementations of it representing typical actions:

- `ON_OVERFLOW_ERROR`: this is the default behavior of the previous two overloads, signalling a `BufferOverflowException`
- `ON_OVERFLOW_DEFAULT`: currently it is the same as `ON_OVERFLOW_ERROR`

- `ON_OVERFLOW_DROP_LATEST` : if an overflow would happen, the current value will be simply ignored and only the old values will be delivered once the downstream requests.
- `ON_OVERFLOW_DROP_OLDEST` : drops the oldest element in the buffer and adds the current value to it.

```
Flowable.range(1, 1_000_000)
          .onBackpressureBuffer(16, () -> { },
              BufferOverflowStrategy.ON_OVERFLOW_DROP_OLDEST)
          .observeOn(Schedulers.computation())
          .subscribe(e -> { }, Throwable::printStackTrace);
```

Note that the last two strategies cause discontinuity in the stream as they drop out elements. In addition, they won't signal `BufferOverflowException`.

## onBackpressureDrop()

Whenever the downstream is not ready to receive values, this operator will drop that elemenet from the sequence. One can think of it as a 0 capacity `onBackpressureBuffer` with strategy `ON_OVERFLOW_DROP_LATEST`.

This operator is useful when one can safely ignore values from a source (such as mouse moves or current GPS location signals) as there will be more up-to-date values later on.

```
component.mouseMoves()
.onBackpressureDrop()
.observeOn(Schedulers.computation(), 1)
.subscribe(event -> compute(event.x, event.y));
```

It may be useful in conjunction with the source operator `interval()`. For example, if one wants to perform some periodic background task but each iteration may last longer than the period, it is safe to drop the excess interval notification as there will be more later on:

```
Flowable.interval(1, TimeUnit.MINUTES)
.onBackpressureDrop()
.observeOn(Schedulers.io())
.doOnNext(e -> networkCall.doStuff())
.subscribe(v -> { }, Throwable::printStackTrace);
```

There exist one overload of this operator: `onBackpressureDrop(Consumer<? super T> onDrop)` where the (shared) action is called with the value being dropped. This variant allows cleaning up the values themselves (e.g., releasing associated resources).

### onBackpressureLatest()

The final operator keeps only the latest value and practically overwrites older, undelivered values. One can think of this as a variant of the `onBackpressureBuffer` with a capacity of 1 and strategy of `ON_OVERFLOW_DROP_OLDEST`.

Unlike `onBackpressureDrop` there is always a value available for consumption if the downstream happened to be lagging behind. This can be useful in some telemetry-like situations where the data may come in some bursty pattern but only the very latest is interesting for processing.

For example, if the user clicks a lot on the screen, we'd still want to react to its latest input.

```
component.mouseClicks()
.onBackpressureLatest()
.observeOn(Schedulers.computation())
.subscribe(event -> compute(event.x, event.y), Throwable::printStackTrace);
```

The use of `onBackpressureDrop` in this case would lead to a situation where the very last click gets dropped and leaves the user wondering why the business logic wasn't executed.

## Creating backpressured datasources

Creating backpressured data sources is the relatively easier task when dealing with backpressure in general because the library already offers static methods on `Flowable` that handle backpressure for the developer. We can distinguish two kinds of factory methods: cold "generators" that either return and generate elements based on downstream demand and hot "pushers" that usually bridge non-reactive and/or non-backpressurable data sources and layer some backpressure handling on top of them.

### just

The most basic backpressure aware source is created via `just`:

```
Flowable.just(1).subscribe(new DisposableSubscriber<Integer>() {
    @Override
    public void onStart() {
        request(0);
    }

    @Override
    public void onNext(Integer v) {
        System.out.println(v);
    }
    // the rest is omitted for brevity
}
```

Since we explicitly don't request in `onStart`, this will not print anything. `just` is great when there is a constant value we'd like to jump-start a sequence.

Unfortunately, `just` is often mistaken for a way to compute something dynamically to be consumed by `Subscribers`:

```
int counter;

int computeValue() {
   return ++counter;
}

Flowable<Integer> o = Flowable.just(computeValue());

o.subscribe(System.out::println);
o.subscribe(System.out::println);
```

Surprising to some, this prints 1 twice instead of printing 1 and 2 respectively. If the call is rewritten, it becomes obvious why it works so:

```
int temp = computeValue();

Flowable<Integer> o = Flowable.just(temp);
```

The `computeValue` is called as part of the main routine and not in response to the subscribers subscribing.

## fromCallable

What people actually need is the method `fromCallable`:

```
Flowable<Integer> o = Flowable.fromCallable(() -> computeValue());
```

Here the `computeValue` is executed only when a subscriber subscribes and for each of them, printing the expected 1 and 2. Naturally, `fromCallable` also properly supports backpressure and won't emit the computed value unless requested. Note however that the computation does happen anyway. In case the computation itself should be delayed until the downstream actually requests, we can use `just` with `map`:

```
Flowable.just("This doesn't matter").map(ignored -> computeValue())...
```

`just` won't emit its constant value until requested when it is mapped to the result of the `computeValue`, still called for each subscriber individually.

## fromArray

If the data is already available as an array of objects, a list of objects or any `Iterable` source, the respective `from` overloads will handle the backpressure and emission of such sources:

```
Flowable.fromArray(1, 2, 3, 4, 5).subscribe(System.out::println);
```

For convenience (and avoiding warnings about generic array creation) there are 2 to 10 argument overloads to `just` that internally delegate to `from`.

The `fromIterable` also gives an interesting opportunity. Many value generation can be expressed in a form of a state-machine. Each requested element triggers a state transition and computation of the returned value.

Writing such state machines as `Iterable`s is somewhat complicated (but still easier than writing an `Flowable` for consuming it) and unlike C#, Java doesn't have any support from the compiler to build such state machines by simply writing classically looking code (with `yield return` and `yield break`). Some libraries offer some help, such as Google Guava's `AbstractIterable` and IxJava's `Ix.generate()` and `Ix.forloop()`. These are by themselves worthy of a full series so let's see some very basic `Iterable` source that repeats some constant value indefinitely:

```
Iterable<Integer> iterable = () -> new Iterator<Integer>() {
    @Override
    public boolean hasNext() {
        return true;
    }

    @Override
    public Integer next() {
        return 1;
    }
};

Flowable.fromIterable(iterable).take(5).subscribe(System.out::println);
```

If we'd consume the `iterator` via classic for-loop, that would result in an infinite loop. Since we build an `Flowable` out of it, we can express our will to consume only the first 5 of it and then stop requesting anything. This is the true power of lazily evaluating and computing inside `Flowable`s.

## generate()

Sometimes, the data source to be converted into the reactive world itself is synchronous (blocking) and pull-like, that is, we have to call some `get` or `read` method to get the next piece of data. One could, of course, turn that into an `Iterable` but when such sources are associated with resources, we may leak those resources if the downstream unsubscribes the sequence before it would end.

To handle such cases, RxJava has the `generate` factory method family.

```
Flowable<Integer> o = Flowable.generate(
    () -> new FileInputStream("data.bin"),
    (inputstream, output) -> {
        try {
            int abyte = inputstream.read();
            if (abyte < 0) {
                output.onComplete();
            } else {
                output.onNext(abyte);
            }
        } catch (IOException ex) {
            output.onError(ex);
        }
        return inputstream;
    },
    inputstream -> {
        try {
            inputstream.close();
        } catch (IOException ex) {
            RxJavaPlugins.onError(ex);
        }
    }
);
```

Generally, `generate` uses 3 callbacks.

The first callbacks allows one to create a per-subscriber state, such as the `FileInputStream` in the example; the file will be opened independently to each individual subscriber.

The second callback takes this state object and provides an output `Observer` whose `onXXX` methods can be called to emit values. This callback is executed as many times as the downstream requested. At each invocation, it has to call `onNext` at most once optionally followed by either `onError` or `onComplete`. In the example we call `onComplete()` if the read byte is negative, indicating and end of file, and call `onError` in case the read throws an `IOException`.

The final callback gets invoked when the downstream unsubscribes (closing the inputstream) or when the previous callback called the terminal methods; it allows freeing up resources. Since not all sources need all these features, the static methods of `Flowable.generate` let's one create instances without them.

Unfortunately, many method calls across the JVM and other libraries throw checked exceptions and need to be wrapped into `try-catch`es as the functional interfaces used by this class don't allow throwing checked exceptions.

Of course, we can imitate other typical sources, such as an unbounded range with it:

```
Flowable.generate(
     () -> 0,
     (current, output) -> {
         output.onNext(current);
         return current + 1;
     },
     e -> { }
);
```

In this setup, the `current` starts out with `0` and next time the lambda is invoked, the parameter `current` now holds `1`.

*(Remark: the 1.x classes `SyncOnSubscribe` and `AsyncOnSubscribe` are no longer available.)*

## create(emitter)

Sometimes, the source to be wrapped into an `Flowable` is already hot (such as mouse moves) or cold but not backpressurable in its API (such as an asynchronous network callback).

To handle such cases, a recent version of RxJava introduced the `create(emitter)` factory method. It takes two parameters:

- a callback that will be called with an instance of the `Emitter<T>` interface for each incoming subscriber,
- a `BackpressureStrategy` enumeration that mandates the developer to specify the backpressure behavior to be applied. It has the usual modes, similar to `onBackpressureXXX` in addition to signalling a `MissingBackpressureException` or simply ignoring such overflow inside it altogether.

Note that it currently doesn't support additional parameters to those backpressure modes. If one needs those customization, using `NONE` as the backpressure mode and applying the relevant `onBackpressureXXX` on the resulting `Flowable` is the way to go.

The first typical case for its use when one wants to interact with a push-based source, such as GUI events. Those APIs feature some form of `addListener`/`removeListener` calls that one can utilize:

```
Flowable.create(emitter -> {
    ActionListener al = e -> {
        emitter.onNext(e);
    };
    button.addActionListener(al);

    emitter.setCancellation(() ->
        button.removeListener(al));

}, BackpressureStrategy.BUFFER);
```

The `Emitter` is relatively straightforward to use; one can call `onNext`, `onError` and `onComplete` on it and the operator handles backpressure and unsubscription management on its own. In addition, if the wrapped API supports cancellation (such as the listener removal in the example), one can use the `setCancellation` (or `setSubscription` for `Subscription`-like resources) to register a cancellation callback that gets invoked when the downstream unsubscribes or the `onError`/`onComplete` is called on the provided `Emitter`instance.

These methods allow only a single resource to be associated with the emitter at a time and setting a new one unsubscribes the old one automatically. If one has to handle multiple resources, create a `CompositeSubscription`, associate it with the emitter and then add further resources to the `CompositeSubscription` itself:

```
Flowable.create(emitter -> {
    CompositeSubscription cs = new CompositeSubscription();

    Worker worker = Schedulers.computation().createWorker();

    ActionListener al = e -> {
        emitter.onNext(e);
    };

    button.addActionListener(al);

    cs.add(worker);
    cs.add(Subscriptions.create(() ->
        button.removeActionListener(al));

    emitter.setSubscription(cs);

}, BackpressureMode.BUFFER);
```

The second scenario usually involves some asynchronous, callback-based API that has to be converted into an `Flowable`.

```
Flowable.create(emitter -> {

    someAPI.remoteCall(new Callback<Data>() {
        @Override
```

```
            public void onSuccess(Data data) {
                emitter.onNext(data);
                emitter.onComplete();
            }

            @Override
            public void onFailure(Exception error) {
                emitter.onError(error);
            }
        });

    }, BackpressureMode.LATEST);
```

In this case, the delegation works the same way. Unfortunately, usually, these classical callback-style APIs don't support cancellation, but if they do, one can setup their cancellation just like in the previoius examples (with perhaps a more involved way though). Note the use of the `LATEST` backpressure mode; if we know there will be only a single value, we don't need the `BUFFER` strategy as it allocates a default 128 element long buffer (that grows as necessary) that is never going to be fully utilized.