Overview of the Flowable Class

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

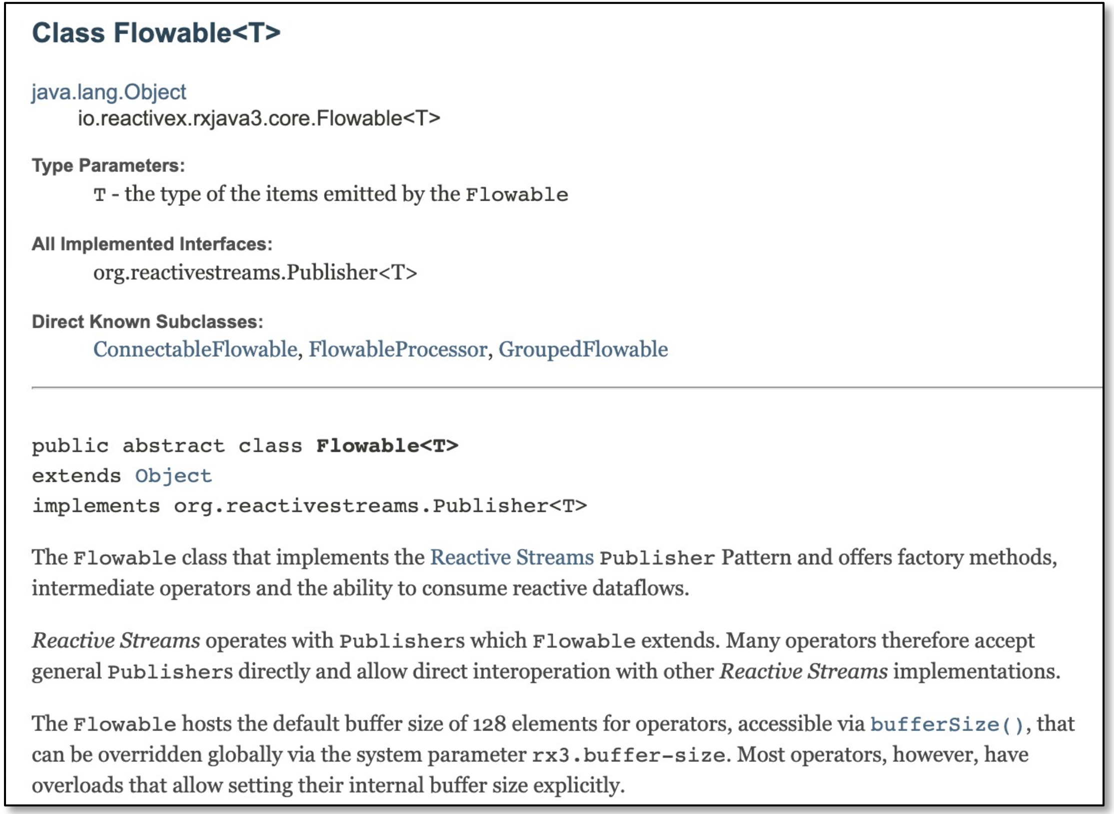
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand the capabilities of the Flowable class



Class Flowable<T>

java.lang.Object
    io.reactivex.rxjava3.core.Flowable<T>

Type Parameters:

`T` - the type of the items emitted by the `Flowable`

All Implemented Interfaces:

org.reactivestreams.Publisher<T>

Direct Known Subclasses:

ConnectableFlowable, FlowableProcessor, GroupedFlowable

```
public abstract class Flowable<T>
extends Object
implements org.reactivestreams.Publisher<T>
```

The `Flowable` class that implements the Reactive Streams `Publisher` Pattern and offers factory methods, intermediate operators and the ability to consume reactive dataflows.

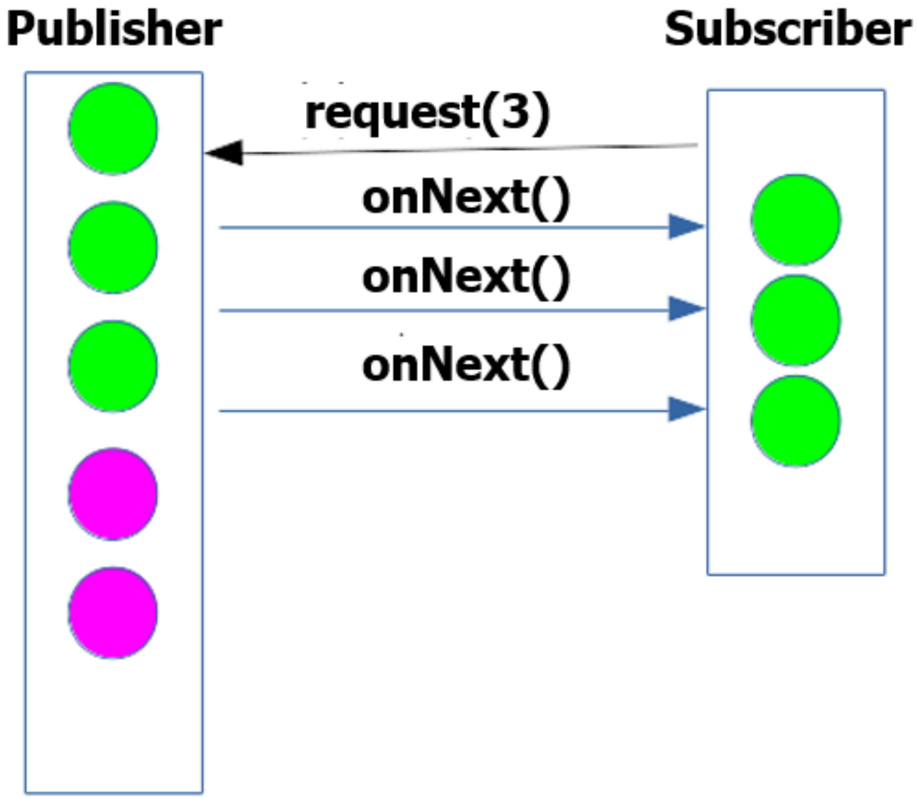
Reactive Streams operates with `Publishers` which `Flowable` extends. Many operators therefore accept general `Publishers` directly and allow direct interoperation with other *Reactive Streams* implementations.

The `Flowable` hosts the default buffer size of 128 elements for operators, accessible via `bufferSize()`, that can be overridden globally via the system parameter `rx3.buffer-size`. Most operators, however, have overloads that allow setting their internal buffer size explicitly.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Flowable.html

Learning Objectives in this Part of the Lesson

- Understand the capabilities of the Flowable class
  - Particularly with respect to its support for *backpressure*
    - Ensures fast publisher(s) don't generate events more quickly than slower subscriber(s) can process them

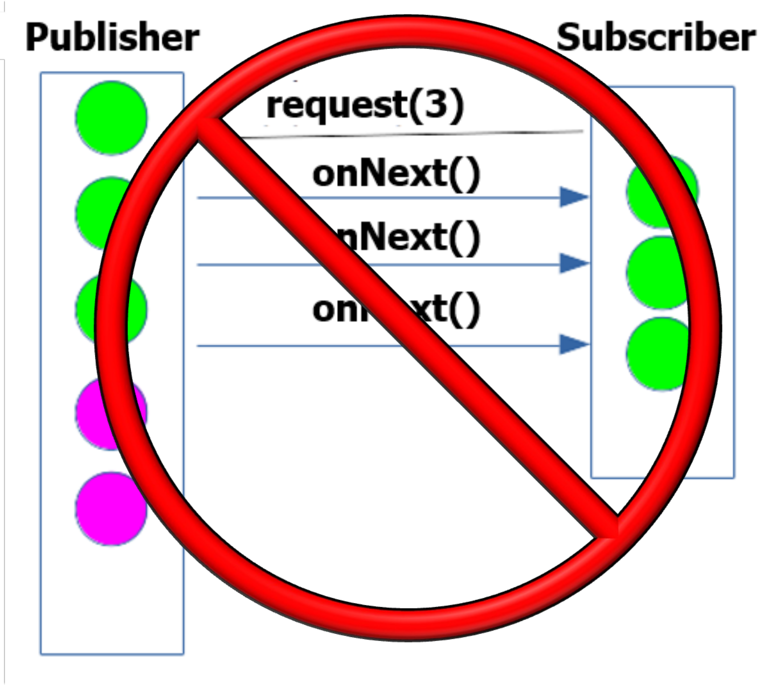


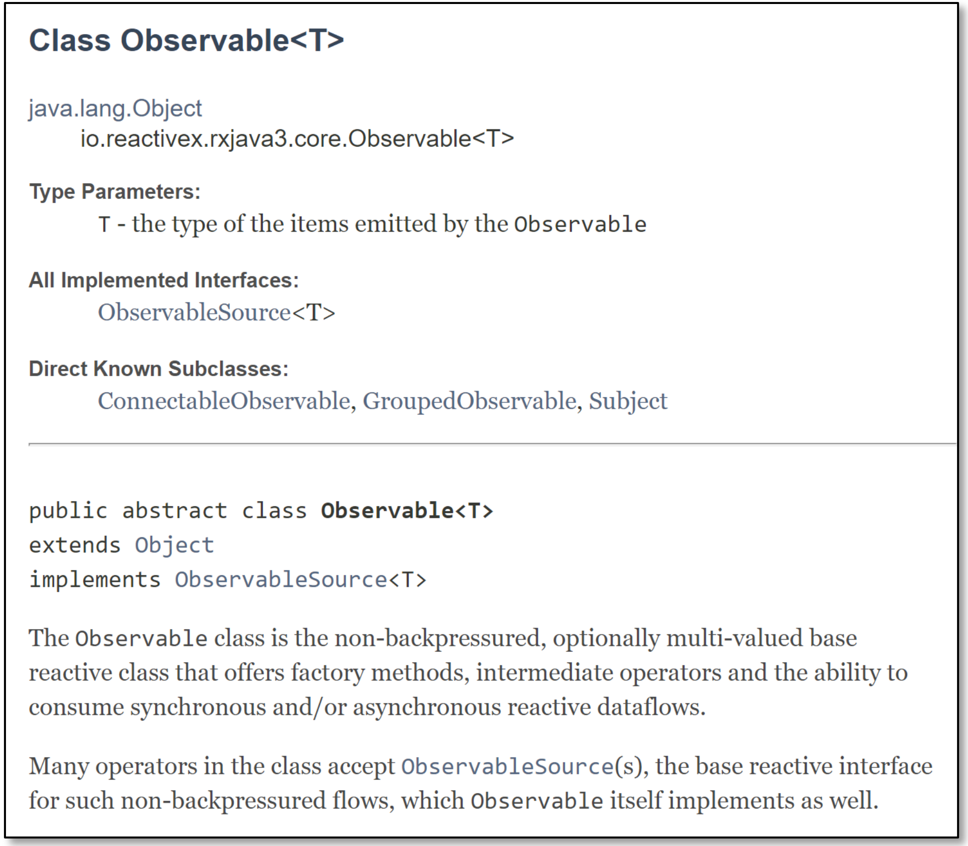
See www.baeldung.com/rxjava-backpressure

Overview of the Flowable Class

Overview of the Flowable Class

- The RxJava Observable class does not support backpressure





Class Observable<T>

java.lang.Object
    io.reactivex.rxjava3.core.Observable<T>

Type Parameters:
    T - the type of the items emitted by the Observable

All Implemented Interfaces:
    ObservableSource<T>

Direct Known Subclasses:
    ConnectableObservable, GroupedObservable, Subject

```
public abstract class Observable<T>
extends Object
implements ObservableSource<T>
```

The Observable class is the non-backpressured, optionally multi-valued base reactive class that offers factory methods, intermediate operators and the ability to consume synchronous and/or asynchronous reactive dataflows.

Many operators in the class accept ObservableSource(s), the base reactive interface for such non-backpressured flows, which Observable itself implements as well.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html

Overview of the Flowable Class

- The RxJava Observable class does not support backpressure
  - It can emit a (potentially endless) stream of elements at a high rate

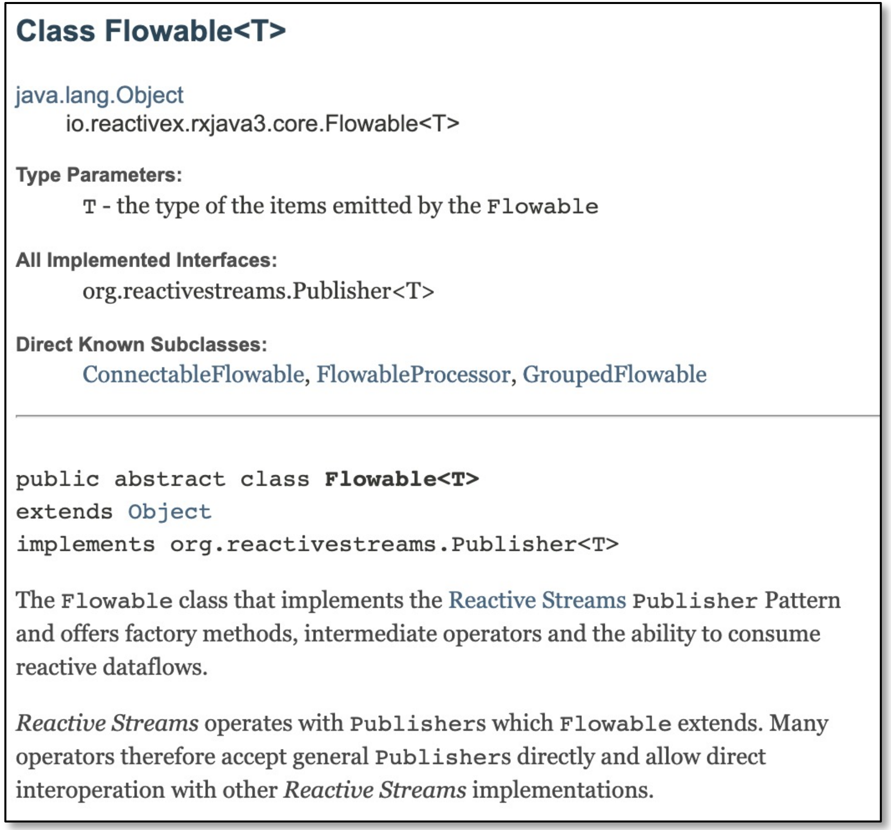
Overview of the Flowable Class

- The RxJava Observable class does not support backpressure
  - It can emit a (potentially endless) stream of elements at a high rate
  - A fast publisher can therefore quickly overwhelm the memory/processing resources of a slower consumer

See www.wideopeneats.com/i-love-lucy-chocolate-factory

Overview of the Flowable Class

- To address this issue the Flowable class was introduced in RxJava 2.x



Class Flowable<T>

java.lang.Object
    io.reactivex.rxjava3.core.Flowable<T>

Type Parameters:

`T` - the type of the items emitted by the `Flowable`

All Implemented Interfaces:

org.reactivestreams.Publisher<T>

Direct Known Subclasses:

ConnectableFlowable, FlowableProcessor, GroupedFlowable

```
public abstract class Flowable<T>
extends Object
implements org.reactivestreams.Publisher<T>
```

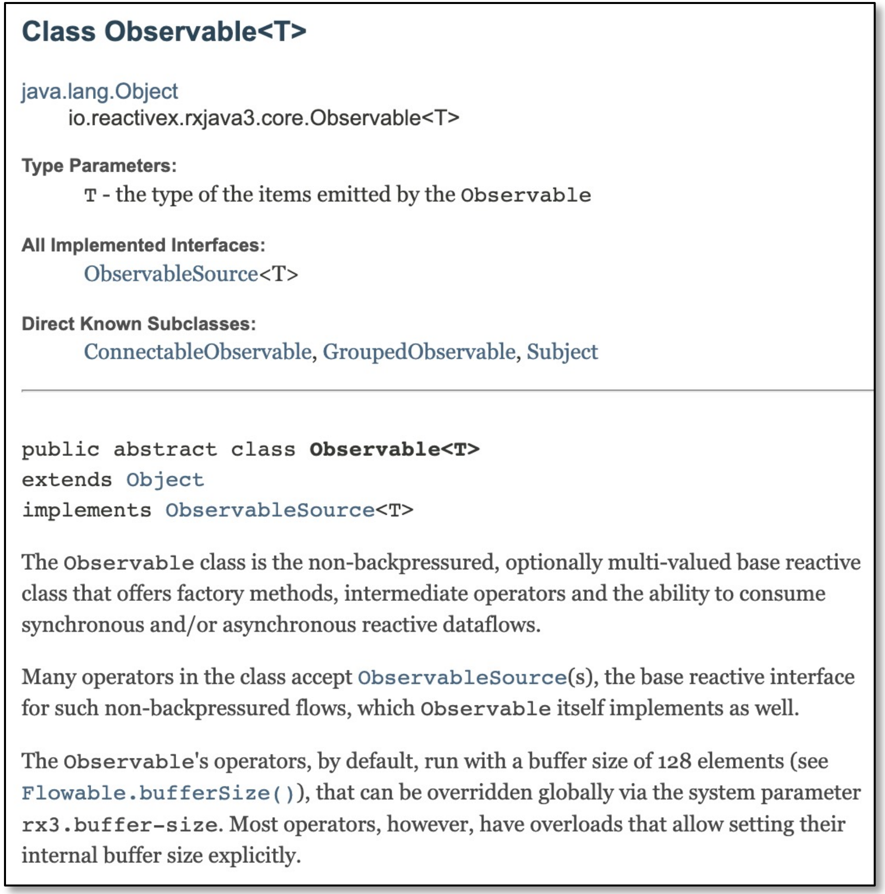
The `Flowable` class that implements the Reactive Streams `Publisher` Pattern and offers factory methods, intermediate operators and the ability to consume reactive dataflows.

Reactive Streams operates with `Publishers` which `Flowable` extends. Many operators therefore accept general `Publishers` directly and allow direct interoperation with other *Reactive Streams* implementations.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Flowable.html

Overview of the Flowable Class

- To address this issue the Flowable class was introduced in RxJava 2.x
  - Most of its operators are the same as the Observable class



Class Observable<T>

java.lang.Object
    io.reactivex.rxjava3.core.Observable<T>

Type Parameters:
    `T` - the type of the items emitted by the `Observable`

All Implemented Interfaces:
    ObservableSource<T>

Direct Known Subclasses:
    ConnectableObservable, GroupedObservable, Subject

```
public abstract class Observable<T>
extends Object
implements ObservableSource<T>
```

The `Observable` class is the non-backpressured, optionally multi-valued base reactive class that offers factory methods, intermediate operators and the ability to consume synchronous and/or asynchronous reactive dataflows.

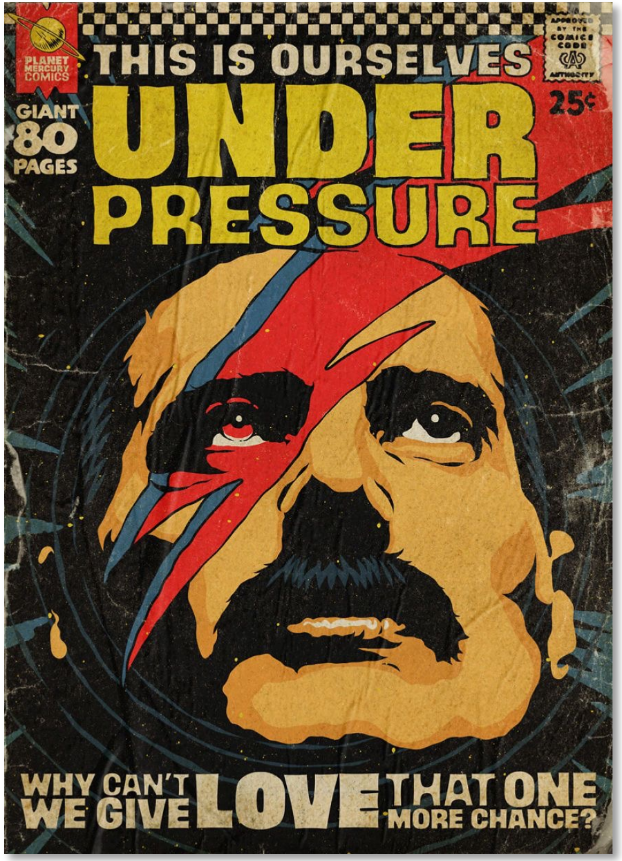
Many operators in the class accept `ObservableSource`(s), the base reactive interface for such non-backpressured flows, which `Observable` itself implements as well.

The `Observable`'s operators, by default, run with a buffer size of 128 elements (see `Flowable.bufferSize()`), that can be overridden globally via the system parameter `rx3.buffer-size`. Most operators, however, have overloads that allow setting their internal buffer size explicitly.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Flowable.html

Overview of the Flowable Class

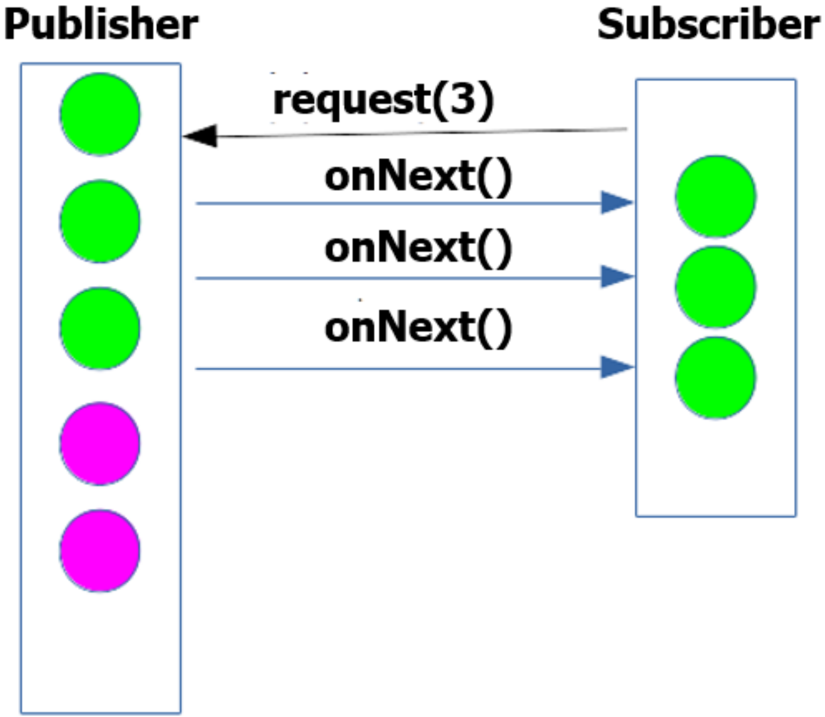
- To address this issue the Flowable class was introduced in RxJava 2.x
  - Most of its operators are the same as the Observable class
  - However, it supports *backpressure*



See medium.com/android-news/rxjava-flowables-what-when-and-how-to-use-it-9f674eb3ecb5

Overview of the Flowable Class

- To address this issue the Flowable class was introduced in RxJava 2.x
  - Most of its operators are the same as the Observable class
  - However, it supports *backpressure*, e.g.
    - Backpressure-aware Subscriber(s) can inform publisher(s) how much data they can consume



See medium.com/android-news/rxjava-flowables-what-when-and-how-to-use-it-9f674eb3ecb5

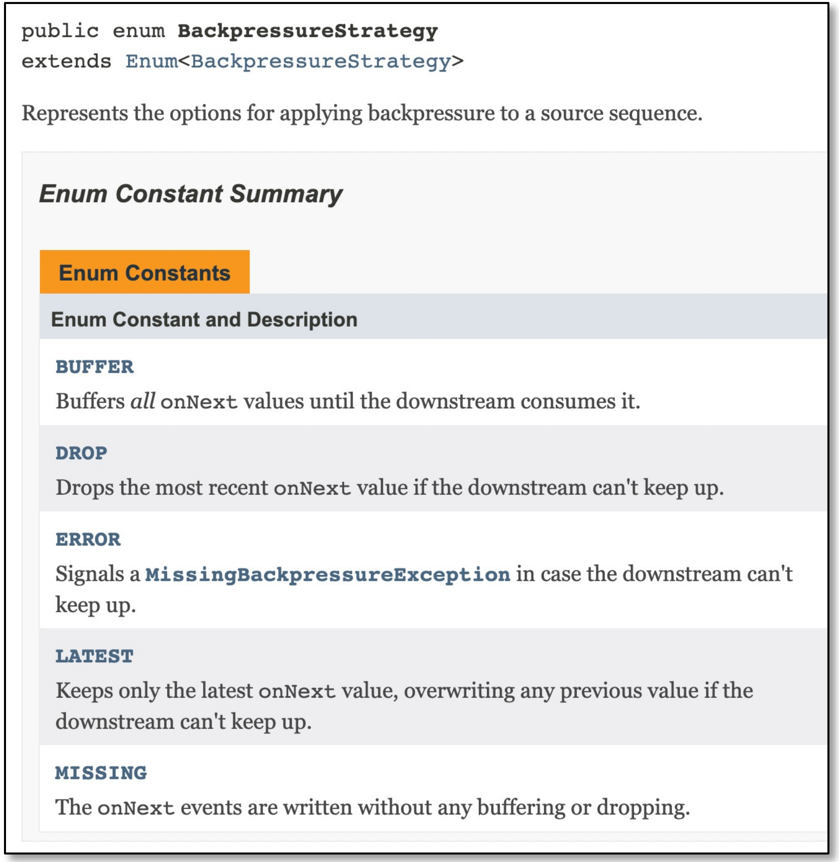
Overview of the Flowable Class

- To address this issue the Flowable class was introduced in RxJava 2.x
  - Most of its operators are the same as the Observable class
  - However, it supports *backpressure*, e.g.
    - Backpressure-aware Subscriber(s) can inform Publisher(s) how much data they can consume
      - i.e., avoid overwhelming memory/processing resources by ensuring flow-controlled Publisher(s) don't generate events faster than Subscriber(s) can consume them

See www.baeldung.com/rxjava-backpressure

Overview of the Flowable Class

- To address this issue the Flowable class was introduced in RxJava 2.x
  - Most of its operators are the same as the Observable class
  - However, it supports *backpressure*, e.g.
    - Backpressure-aware Subscriber(s) can inform Publisher(s) how much data they can consume
    - Non-backpressure-aware Subscriber(s) can apply a strategy if they can't keep up with faster Publisher(s)



```
public enum BackpressureStrategy
extends Enum<BackpressureStrategy>
```

Represents the options for applying backpressure to a source sequence.

Enum Constant Summary

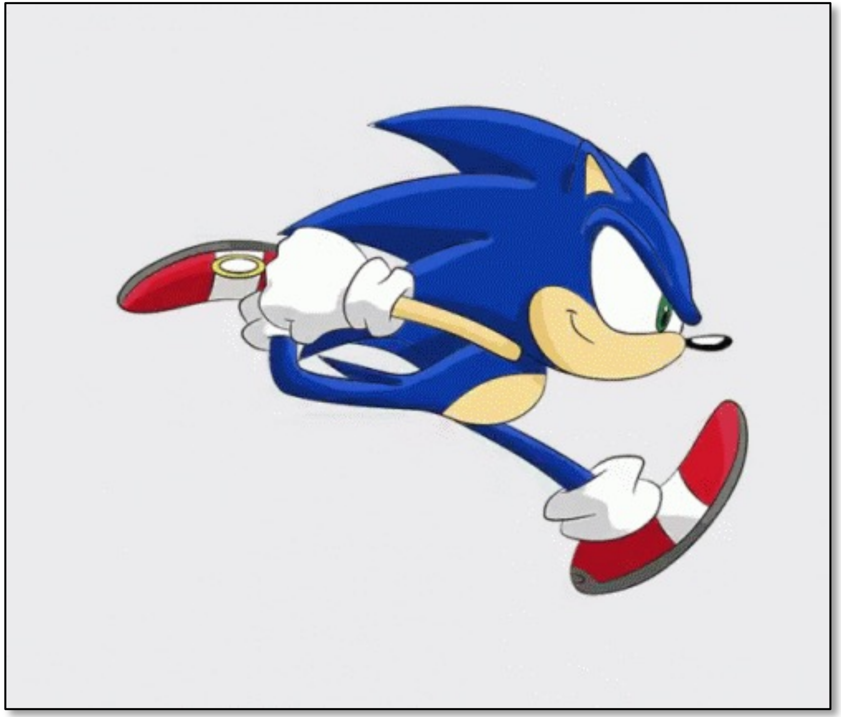
Enum Constants

Enum Constant and Description
BUFFER Buffers *all* `onNext` values until the downstream consumes it.
DROP Drops the most recent `onNext` value if the downstream can't keep up.
ERROR Signals a **MissingBackpressureException** in case the downstream can't keep up.
LATEST Keeps only the latest `onNext` value, overwriting any previous value if the downstream can't keep up.
MISSING The `onNext` events are written without any buffering or dropping.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/BackpressureStrategy.html

Overview of the Flowable Class

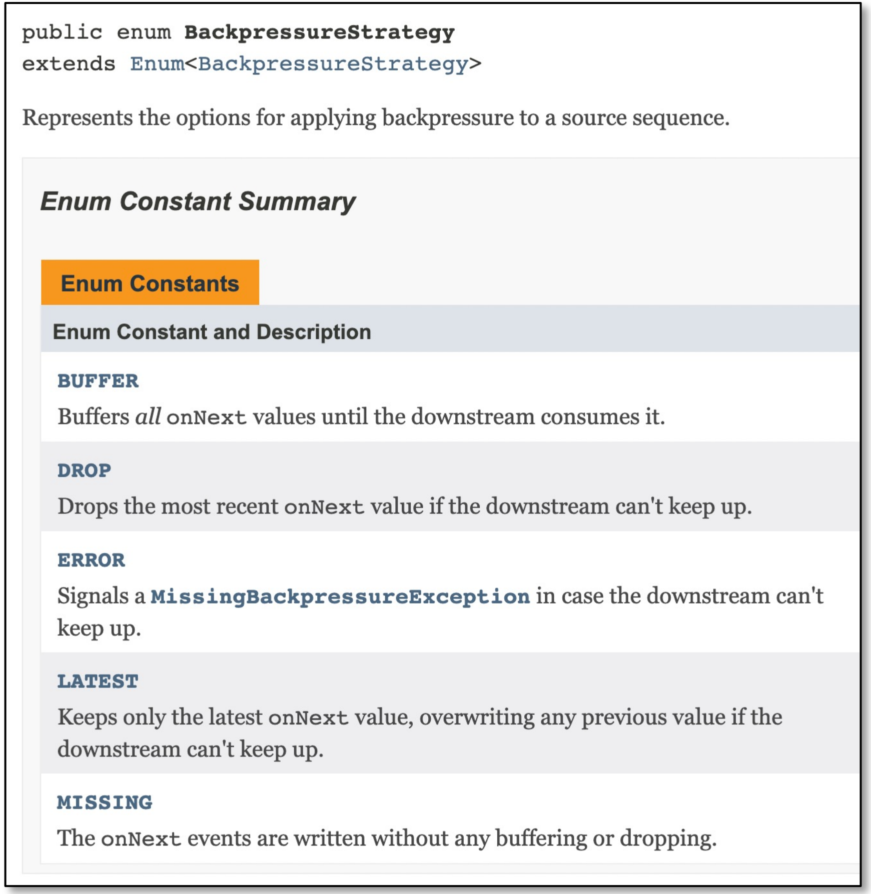
- To address this issue the Flowable class was introduced in RxJava 2.x
  - Most of its operators are the same as the Observable class
  - However, it supports *backpressure*, e.g.
    - Backpressure-aware Subscriber(s) can inform Publisher(s) how much data they can consume
    - Non-backpressure-aware Subscriber(s) can apply a strategy if they can't keep up with faster Publisher(s)
      - i.e., non-flow-controlled Publisher(s)



Overview of Back pressure Strategies

Overview of Backpressure Strategies

- Backpressure strategies say how to handle emitted items that can't be processed as fast as they're received



```
public enum BackpressureStrategy
extends Enum<BackpressureStrategy>
```

Represents the options for applying backpressure to a source sequence.

Enum Constant Summary

Enum Constants

Enum Constant and Description

BUFFER
Buffers *all* `onNext` values until the downstream consumes it.

DROP
Drops the most recent `onNext` value if the downstream can't keep up.

ERROR
Signals a **`MissingBackpressureException`** in case the downstream can't keep up.

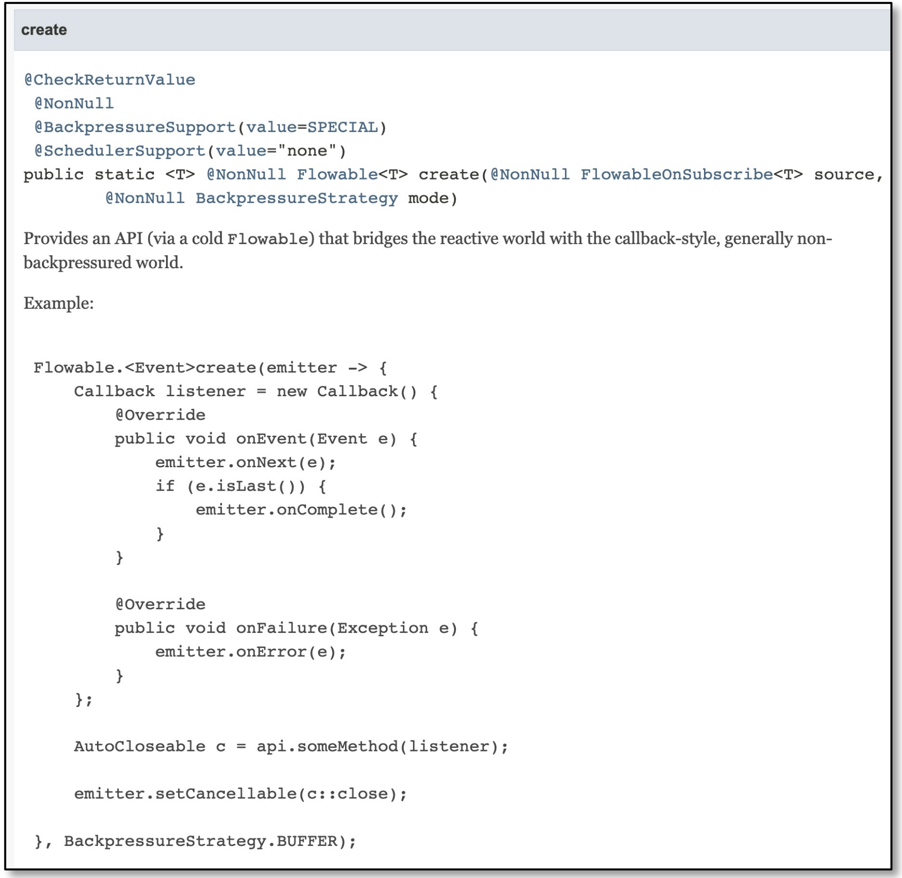
LATEST
Keeps only the latest `onNext` value, overwriting any previous value if the downstream can't keep up.

MISSING
The `onNext` events are written without any buffering or dropping.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/BackpressureStrategy.html

Overview of Backpressure Strategies

- Backpressure strategies say how to handle emitted items that can’t be processed as fast as they’re received
  - These strategies can be provided via the Flowable.create() operator



create

```
@CheckReturnValue
 @NonNull
 @BackpressureSupport(value=SPECIAL)
 @SchedulerSupport(value="none")
public static <T> @NonNull Flowable<T> create(@NonNull FlowableOnSubscribe<T> source,
        @NonNull BackpressureStrategy mode)
```

Provides an API (via a cold `Flowable`) that bridges the reactive world with the callback-style, generally non-backpressured world.

Example:

```
 Flowable.<Event>create(emitter -> {
     Callback listener = new Callback() {
         @Override
         public void onEvent(Event e) {
             emitter.onNext(e);
             if (e.isLast()) {
                 emitter.onComplete();
             }
         }

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

     AutoCloseable c = api.someMethod(listener);

     emitter.setCancellable(c::close);

 }, BackpressureStrategy.BUFFER);
```

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Flowable.html#create

Overview of Backpressure Strategies

- Backpressure strategies say how to handle emitted items that can't be processed as fast as they're received
  - These strategies can be provided via the Flowable.create() operator
    - Specify the backpressure mode to apply if the Subscriber can't keep up with the Publisher

```
Flowable.create
  (emitter -> { Flowable
     .range(1, count)
     .subscribe(___ ->
        emitter.onNext(random
                 .nextInt(max)),
        emitter::onError,
        emitter::onComplete)
   },

   BackpressureStrategy.DROP)

   ...
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/Flowable/ex1

Overview of Backpressure Strategies

- Backpressure strategies say how to handle emitted items that can’t be processed as fast as they’re received
  - These strategies can be provided via the Flowable.create() operator
    - Specify the backpressure mode to apply if the Subscriber can’t keep up with the Publisher

```
Flowable.create
   (emitter -> { Flowable
       .range(1, count)
       .subscribe(___ ->
          emitter.onNext(random
                   .nextInt(max)),
          emitter::onError,
          emitter::onComplete)
    },

    BackpressureStrategy.DROP)

    ...
```

Rapidly emit a stream of random Integer objects in one fell swoop

See github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/Flowable/ex1

Overview of Backpressure Strategies

- Backpressure strategies say how to handle emitted items that can't be processed as fast as they're received
  - These strategies can be provided via the Flowable.create() operator
    - Specify the backpressure mode to apply if the Subscriber can't keep up with the Publisher

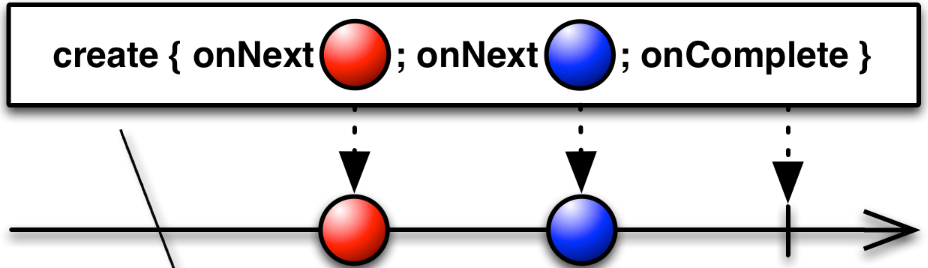
```
Flowable.create
  (emitter -> { Flowable
     .range(1, count)
     .subscribe(___ ->
        emitter.onNext(random
                 .nextInt(max)),
        emitter::onError,
        emitter::onComplete)
   },

   BackpressureStrategy.DROP)

   ...
```

Ignore all streamed items that can't be processed immediately

See github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/Flowable/ex1

Overview of Backpressure Strategies

- Backpressure strategies say how to handle emitted items that can't be processed as fast as they're received
  - These strategies can be provided via the Flowable.create() operator
    - Specify the backpressure mode to apply if the Subscriber can't keep up with the Publisher
    - This operator is different than Observable.create()

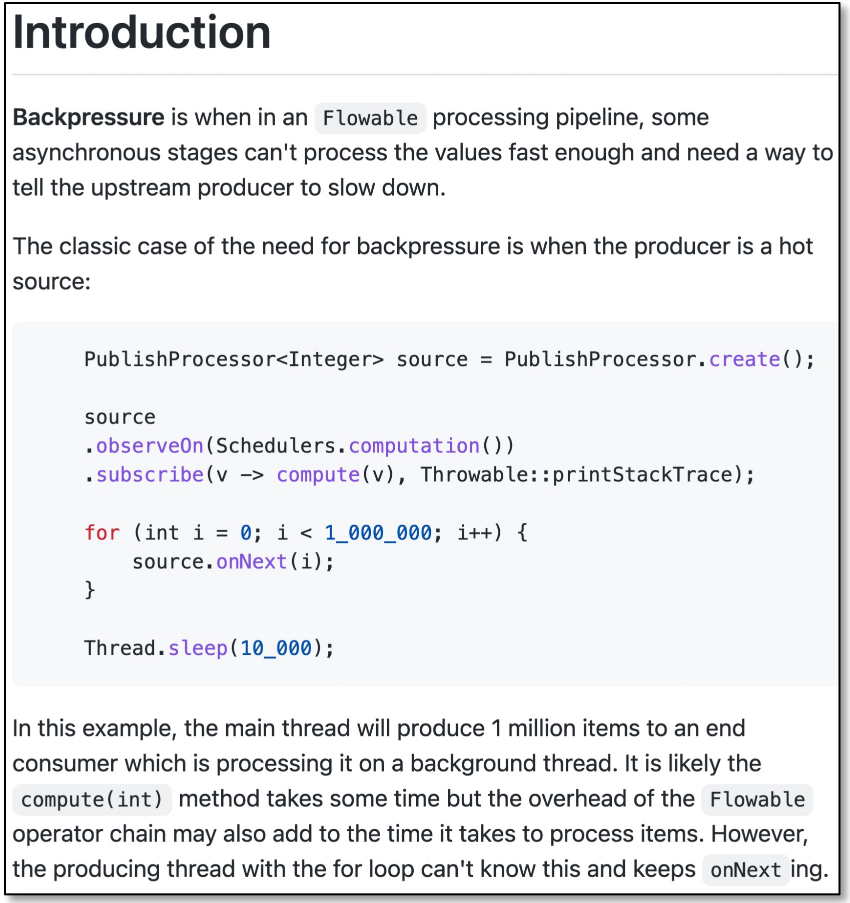


Observerable.create() does not support backpressure

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#create

Overview of Backpressure Strategies

- Backpressure strategies say how to handle emitted items that can't be processed as fast as they're received
  - These strategies can be provided via the Flowable.create() operator
  - They can also be provided via various Flowable operators



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.

See github.com/ReactiveX/RxJava/wiki/Backpressure-(2.0)

Overview of Backpressure Strategies

- Backpressure strategies say how to handle emitted items that can't be processed as fast as they're received
  - These strategies can be provided via the Flowable.create() operator
  - They can also be provided via various Flowable operators
    - onBackpressureDrop()
      - Ignore all streamed items that can't be processed until down stream can accept more of them

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

See github.com/ReactiveX/RxJava/wiki/Backpressure-(2.0)#onbackpressuredrop

Overview of Backpressure Strategies

- Backpressure strategies say how to handle emitted items that can't be processed as fast as they're received
  - These strategies can be provided via the Flowable.create() operator
  - They can also be provided via various Flowable operators
    - onBackpressureBuffer()
      - Creates a bounded or unbounded buffer that holds the emitted items that couldn't be processed by the downstream

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

See github.com/ReactiveX/RxJava/wiki/Backpressure-(2.0)#onbackpressurebuffer

Overview of Backpressure Strategies

- Backpressure strategies say how to handle emitted items that can't be processed as fast as they're received
  - These strategies can be provided via the Flowable.create() operator
  - They can also be provided via various Flowable operators
    - onBackpressureLatest()
      - Like the DROP strategy, but it keeps the last emitted item

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

See github.com/ReactiveX/RxJava/wiki/Backpressure-(2.0)#onbackpressurelatest

End of Overview of the Flowable Class