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 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](http://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
public abstract class Observable<T>
    extends Object
    implements ObservableSource<T>
```

- **Overview**
  - The `Observable` class is the non-back pressured, 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-back pressured flows, which `Observable` itself implements as well.

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](http://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
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`

```java
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](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
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

The Observable class is the non-backpressed, 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-backpressed 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](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
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)

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

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

Represents the options for applying backpressure to a source sequence.

<table>
<thead>
<tr>
<th>Enum Constant Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum Constant and Description</td>
</tr>
<tr>
<td>BUFFER</td>
</tr>
<tr>
<td>DROP</td>
</tr>
<tr>
<td>ERROR</td>
</tr>
<tr>
<td>LATEST</td>
</tr>
<tr>
<td>MISSING</td>
</tr>
</tbody>
</table>
```

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.

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

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

  BackpressureStrategy.DROP)

...
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.

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

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/Flowable/ex1](https://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.

Example:

```java
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/douglasraigschmidt/LiveLessons/tree/master/Reactive/Flowable/ex1](https://github.com/douglasraigschmidt/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()

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:

```java
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)](https://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 downstream can accept more of them.

See [github.com/ReactiveX/RxJava/wiki/Backpressure-(2.0)#onbackpressuredrop](https://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.

```java
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](https://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