This page shows operators that perform mathematical or other operations over an entire sequence of items emitted by an `Observable` or `Flowable`. Because these operations must wait for the source `Observable` / `Flowable` to complete emitting items before they can construct their own emissions (and must usually buffer these items), these operators are dangerous to use on `Observable`s and `Flowable`s that may have very long or infinite sequences.

Outline

Mathematical Operators

> *The operators in this section are part of the `RxJava2Extensions` project. You have to add the `rxjava2-extensions` module as a dependency to your project. It can be found at http://search.maven.org.*

> *Note that unlike the standard RxJava aggregator operators, these mathematical operators return `Observable` and `Flowable` instead of the `Single` or `Maybe`.*

The examples below assume that the `MathObservable` and `MathFlowable` classes are imported from the `rxjava2-extensions` module:

```
import hu.akarnokd.rxjava2.math.MathObservable;
import hu.akarnokd.rxjava2.math.MathFlowable;
```

averageDouble

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/average.html

Calculates the average of `Number` s emitted by an `Observable` and emits this average as a `Double` .

averageDouble example

```
Observable<Integer> numbers = Observable.just(1, 2, 3);
MathObservable.averageDouble(numbers).subscribe((Double avg) ->
System.out.println(avg));

// prints 2.0
```

averageFloat

Available in:  `Flowable` ,  `Observable` ,  `Maybe` ,  `Single` ,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/average.html

Calculates the average of `Number` s emitted by an `Observable` and emits this average as a `Float` .

averageFloat example

```
Observable<Integer> numbers = Observable.just(1, 2, 3);
MathObservable.averageFloat(numbers).subscribe((Float avg) ->
System.out.println(avg));

// prints 2.0
```

max

Available in:  `Flowable` ,  `Observable` ,  `Maybe` ,  `Single` ,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/max.html

Emits the maximum value emitted by a source `Observable` . A `Comparator` can be specified that will be used to compare the elements emitted by the `Observable` .

max example

```
Observable<Integer> numbers = Observable.just(4, 9, 5);
MathObservable.max(numbers).subscribe(System.out::println);

// prints 9
```

The following example specifies a `Comparator` to find the longest `String` in the source `Observable` :

```
final Observable<String> names = Observable.just("Kirk", "Spock", "Chekov", "Sulu");
MathObservable.max(names, Comparator.comparingInt(String::length))
        .subscribe(System.out::println);

// prints Chekov
```

min

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/min.html

Emits the minimum value emitted by a source `Observable`. A `Comparator` can be specified that will be used to compare the elements emitted by the `Observable`.

min example

```
Observable<Integer> numbers = Observable.just(4, 9, 5);
MathObservable.min(numbers).subscribe(System.out::println);

// prints 4
```

sumDouble

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/sum.html

Adds the `Double`s emitted by an `Observable` and emits this sum.

sumDouble example

```
Observable<Double> numbers = Observable.just(1.0, 2.0, 3.0);
MathObservable.sumDouble(numbers).subscribe((Double sum) ->
System.out.println(sum));

// prints 6.0
```

sumFloat

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/sum.html

Adds the `Float`s emitted by an `Observable` and emits this sum.

sumFloat example

```
Observable<Float> numbers = Observable.just(1.0F, 2.0F, 3.0F);
MathObservable.sumFloat(numbers).subscribe((Float sum) -> System.out.println(sum));

// prints 6.0
```

sumInt

Available in: `Flowable`, `Observable`, `Maybe`, `Single`, `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/sum.html

Adds the `Integer`s emitted by an `Observable` and emits this sum.

sumInt example

```
Observable<Integer> numbers = Observable.range(1, 100);
MathObservable.sumInt(numbers).subscribe((Integer sum) -> System.out.println(sum));

// prints 5050
```

sumLong

Available in: `Flowable`, `Observable`, `Maybe`, `Single`, `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/sum.html

Adds the `Long`s emitted by an `Observable` and emits this sum.

sumLong example

```
Observable<Long> numbers = Observable.rangeLong(1L, 100L);
MathObservable.sumLong(numbers).subscribe((Long sum) -> System.out.println(sum));

// prints 5050
```

Standard Aggregate Operators

> *Note that these standard aggregate operators return a `Single` or `Maybe` because the number of output items is always know to be at most one.*

count

Available in: `Flowable`, `Observable`, `Maybe`, `Single`, `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/count.html

Counts the number of items emitted by an `Observable` and emits this count as a `Long`.

count example

```
Observable.just(1, 2, 3).count().subscribe(System.out::println);

// prints 3
```

reduce

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/reduce.html

Apply a function to each emitted item, sequentially, and emit only the final accumulated value.

reduce example

```
Observable.range(1, 5)
    .reduce((product, x) -> product * x)
    .subscribe(System.out::println);

// prints 120
```

reduceWith

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/reduce.html

Apply a function to each emitted item, sequentially, and emit only the final accumulated value.

reduceWith example

```
Observable.just(1, 2, 2, 3, 4, 4, 4, 5)
        .reduceWith(TreeSet::new, (set, x) -> {
            set.add(x);
            return set;
        })
        .subscribe(System.out::println);

// prints [1, 2, 3, 4, 5]
```

collect

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/reduce.html

Collect items emitted by the source `Observable` into a single mutable data structure and return an `Observable` that emits this structure.

collect example

```
Observable.just("Kirk", "Spock", "Chekov", "Sulu")
        .collect(() -> new StringJoiner(" \uD83D\uDD96 "), StringJoiner::add)
        .map(StringJoiner::toString)
        .subscribe(System.out::println);

// prints Kirk 🖖 Spock 🖖 Chekov 🖖 Sulu
```

collectInto

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/reduce.html

Collect items emitted by the source `Observable` into a single mutable data structure and return an `Observable` that emits this structure.

collectInto example

Note: the mutable value that will collect the items (here the `StringBuilder`) will be shared between multiple subscribers.

```
Observable.just('R', 'x', 'J', 'a', 'v', 'a')
        .collectInto(new StringBuilder(), StringBuilder::append)
        .map(StringBuilder::toString)
        .subscribe(System.out::println);

// prints RxJava
```

toList

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/to.html

Collect all items from an `Observable` and emit them as a single `List`.

toList example

```
Observable.just(2, 1, 3)
        .toList()
        .subscribe(System.out::println);

// prints [2, 1, 3]
```

toSortedList

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/to.html

Collect all items from an `Observable` and emit them as a single, sorted `List`.

toSortedList example

```
Observable.just(2, 1, 3)
        .toSortedList(Comparator.reverseOrder())
        .subscribe(System.out::println);

// prints [3, 2, 1]
```

toMap

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/to.html

Convert the sequence of items emitted by an `Observable` into a `Map` keyed by a specified key function.

toMap example

```
Observable.just(1, 2, 3, 4)
        .toMap((x) -> {
            // defines the key in the Map
            return x;
        }, (x) -> {
            // defines the value that is mapped to the key
            return (x % 2 == 0) ? "even" : "odd";
        })
        .subscribe(System.out::println);

// prints {1=odd, 2=even, 3=odd, 4=even}
```

toMultimap

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/to.html

Convert the sequence of items emitted by an `Observable` into a `Collection` that is also a `Map` keyed by a specified key function.

toMultimap example

```
Observable.just(1, 2, 3, 4)
        .toMultimap((x) -> {
            // defines the key in the Map
            return (x % 2 == 0) ? "even" : "odd";
        }, (x) -> {
            // defines the value that is mapped to the key
            return x;
        })
        .subscribe(System.out::println);

// prints {even=[2, 4], odd=[1, 3]}
```