This page shows operators you can use to filter and select items emitted by reactive sources, such as `Observable`s.

Outline

debounce

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

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

Drops items emitted by a reactive source that are followed by newer items before the given timeout value expires. The timer resets on each emission.

This operator keeps track of the most recent emitted item, and emits this item only when enough time has passed without the source emitting any other items.

debounce example

```
// Diagram:
// -A--------------B----C-D------------------E-|---->
```

```
//  a---------1s
//                  b---------1s
//                       c---------1s
//                         d---------1s
//                                              e-|---->
// -----------A---------------------D-----------E-|-->

Observable<String> source = Observable.create(emitter -> {
    emitter.onNext("A");

    Thread.sleep(1_500);
    emitter.onNext("B");

    Thread.sleep(500);
    emitter.onNext("C");

    Thread.sleep(250);
    emitter.onNext("D");

    Thread.sleep(2_000);
    emitter.onNext("E");
    emitter.onComplete();
});

source.subscribeOn(Schedulers.io())
        .debounce(1, TimeUnit.SECONDS)
        .blockingSubscribe(
                item -> System.out.println("onNext: " + item),
                Throwable::printStackTrace,
                () -> System.out.println("onComplete"));

// prints:
// onNext: A
// onNext: D
// onNext: E
// onComplete
```

distinct

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

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

Filters a reactive source by only emitting items that are distinct by comparison from previous items. A `io.reactivex.functions.Function` can be specified that projects each item emitted by the source into a new value that will be used for comparison with previous projected values.

distinct example

```
Observable.just(2, 3, 4, 4, 2, 1)
        .distinct()
        .subscribe(System.out::println);

// prints:
// 2
// 3
// 4
// 1
```

distinctUntilChanged

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

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

Filters a reactive source by only emitting items that are distinct by comparison from their immediate predecessors. A `io.reactivex.functions.Function` can be specified that projects each item emitted by the source into a new value that will be used for comparison with previous projected values. Alternatively, a `io.reactivex.functions.BiPredicate` can be specified that is used as the comparator function to compare immediate predecessors with each other.

distinctUntilChanged example

```
Observable.just(1, 1, 2, 1, 2, 3, 3, 4)
        .distinctUntilChanged()
        .subscribe(System.out::println);

// prints:
// 1
// 2
// 1
// 2
// 3
// 4
```

elementAt

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

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

Emits the single item at the specified zero-based index in a sequence of emissions from a reactive source. A default item can be specified that will be emitted if the specified index is not within the sequence.

elementAt example

```
Observable<Long> source = Observable.<Long, Long>generate(() -> 1L, (state, emitter)
-> {
    emitter.onNext(state);

    return state + 1L;
}).scan((product, x) -> product * x);

Maybe<Long> element = source.elementAt(5);
element.subscribe(System.out::println);

// prints 720
```

elementAtOrError

    

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

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

Emits the single item at the specified zero-based index in a sequence of emissions from a reactive source, or signals a `java.util.NoSuchElementException` if the specified index is not within the sequence.

elementAtOrError example

```
Observable<String> source = Observable.just("Kirk", "Spock", "Chekov", "Sulu");
Single<String> element = source.elementAtOrError(4);

element.subscribe(
    name -> System.out.println("onSuccess will not be printed!"),
    error -> System.out.println("onError: " + error));

// prints:
// onError: java.util.NoSuchElementException
```

filter

    

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

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

Filters items emitted by a reactive source by only emitting those that satisfy a specified predicate.

filter example

```
Observable.just(1, 2, 3, 4, 5, 6)
        .filter(x -> x % 2 == 0)
        .subscribe(System.out::println);
```

```
// prints:
// 2
// 4
// 6
```

first

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

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

Emits only the first item emitted by a reactive source, or emits the given default item if the source completes without emitting an item. This differs from `firstElement` in that this operator returns a `Single` whereas `firstElement` returns a `Maybe`.

first example

```
Observable<String> source = Observable.just("A", "B", "C");
Single<String> firstOrDefault = source.first("D");

firstOrDefault.subscribe(System.out::println);

// prints A
```

firstElement

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

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

Emits only the first item emitted by a reactive source, or just completes if the source completes without emitting an item. This differs from `first` in that this operator returns a `Maybe` whereas `first` returns a `Single`.

firstElement example

```
Observable<String> source = Observable.just("A", "B", "C");
Maybe<String> first = source.firstElement();

first.subscribe(System.out::println);

// prints A
```

firstOrError

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

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

Emits only the first item emitted by a reactive source, or signals a `java.util.NoSuchElementException` if the source completes without emitting an item.

firstOrError example

```
Observable<String> emptySource = Observable.empty();
Single<String> firstOrError = emptySource.firstOrError();

firstOrError.subscribe(
        element -> System.out.println("onSuccess will not be printed!"),
        error -> System.out.println("onError: " + error));

// prints:
// onError: java.util.NoSuchElementException
```

ignoreElement

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

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

Ignores the single item emitted by a `Single` or `Maybe` source, and returns a `Completable` that signals only the error or completion event from the the source.

ignoreElement example

```
Single<Long> source = Single.timer(1, TimeUnit.SECONDS);
Completable completable = source.ignoreElement();

completable.doOnComplete(() -> System.out.println("Done!"))
        .blockingAwait();

// prints (after 1 second):
// Done!
```

ignoreElements

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

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

Ignores all items from the `Observable` or `Flowable` source, and returns a `Completable` that signals only the error or completion event from the source.

ignoreElements example

```
Observable<Long> source = Observable.intervalRange(1, 5, 1, 1, TimeUnit.SECONDS);
Completable completable = source.ignoreElements();

completable.doOnComplete(() -> System.out.println("Done!"))
        .blockingAwait();

// prints (after 5 seconds):
// Done!
```

last

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

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

Emits only the last item emitted by a reactive source, or emits the given default item if the source completes without emitting an item. This differs from `lastElement` in that this operator returns a `Single` whereas `lastElement` returns a `Maybe` .

last example

```
Observable<String> source = Observable.just("A", "B", "C");
Single<String> lastOrDefault = source.last("D");

lastOrDefault.subscribe(System.out::println);

// prints C
```

lastElement

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

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

Emits only the last item emitted by a reactive source, or just completes if the source completes without emitting an item. This differs from `last` in that this operator returns a `Maybe` whereas `last` returns a `Single` .

lastElement example

```
Observable<String> source = Observable.just("A", "B", "C");
Maybe<String> last = source.lastElement();
```

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

// prints C
```

lastOrError

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

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

Emits only the last item emitted by a reactive source, or signals a `java.util.NoSuchElementException` if the source completes without emitting an item.

lastOrError example

```
Observable<String> emptySource = Observable.empty();
Single<String> lastOrError = emptySource.lastOrError();

lastOrError.subscribe(
        element -> System.out.println("onSuccess will not be printed!"),
        error -> System.out.println("onError: " + error));

// prints:
// onError: java.util.NoSuchElementException
```

ofType

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

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

Filters items emitted by a reactive source by only emitting those of the specified type.

ofType example

```
Observable<Number> numbers = Observable.just(1, 4.0, 3, 2.71, 2f, 7);
Observable<Integer> integers = numbers.ofType(Integer.class);

integers.subscribe((Integer x) -> System.out.println(x));

// prints:
// 1
// 3
// 7
```

sample

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

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

Filters items emitted by a reactive source by only emitting the most recently emitted item within periodic time intervals.

sample example

```
// Diagram:
// -A----B-C-------D-----E-|-->
// -0s-----c--1s---d----2s-|-->
// -----------C---------D--|-->

Observable<String> source = Observable.create(emitter -> {
    emitter.onNext("A");

    Thread.sleep(500);
    emitter.onNext("B");

    Thread.sleep(200);
    emitter.onNext("C");

    Thread.sleep(800);
    emitter.onNext("D");

    Thread.sleep(600);
    emitter.onNext("E");
    emitter.onComplete();
});

source.subscribeOn(Schedulers.io())
        .sample(1, TimeUnit.SECONDS)
        .blockingSubscribe(
                item -> System.out.println("onNext: " + item),
                Throwable::printStackTrace,
                () -> System.out.println("onComplete"));

// prints:
// onNext: C
// onNext: D
// onComplete
```

skip

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

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

Drops the first *n* items emitted by a reactive source, and emits the remaining items.

skip example

```
Observable<Integer> source = Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

source.skip(4)
    .subscribe(System.out::println);

// prints:
// 5
// 6
// 7
// 8
// 9
// 10
```

skipLast

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

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

Drops the last *n* items emitted by a reactive source, and emits the remaining items.

skipLast example

```
Observable<Integer> source = Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

source.skipLast(4)
    .subscribe(System.out::println);

// prints:
// 1
// 2
// 3
// 4
// 5
// 6
```

take

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

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

Emits only the first *n* items emitted by a reactive source.

take example

```
Observable<Integer> source = Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

source.take(4)
    .subscribe(System.out::println);

// prints:
// 1
// 2
// 3
// 4
```

takeLast

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

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

Emits only the last *n* items emitted by a reactive source.

takeLast example

```
Observable<Integer> source = Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

source.takeLast(4)
    .subscribe(System.out::println);

// prints:
// 7
// 8
// 9
// 10
```

throttleFirst

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

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

Emits only the first item emitted by a reactive source during sequential time windows of a specified duration.

throttleFirst example

```
// Diagram:
// -A----B-C-------D-----E-|-->
//  a---------1s
//                 d-------|-->
// -A--------------D-------|-->

Observable<String> source = Observable.create(emitter -> {
    emitter.onNext("A");

    Thread.sleep(500);
    emitter.onNext("B");

    Thread.sleep(200);
    emitter.onNext("C");

    Thread.sleep(800);
    emitter.onNext("D");

    Thread.sleep(600);
    emitter.onNext("E");
    emitter.onComplete();
});

source.subscribeOn(Schedulers.io())
        .throttleFirst(1, TimeUnit.SECONDS)
        .blockingSubscribe(
                item -> System.out.println("onNext: " + item),
                Throwable::printStackTrace,
                () -> System.out.println("onComplete"));

// prints:
// onNext: A
// onNext: D
// onComplete
```

throttleLast

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

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

Emits only the last item emitted by a reactive source during sequential time windows of a specified duration.

throttleLast example

```
// Diagram:
// -A----B-C-------D-----E-|-->
// -0s-----c--1s---d----2s-|-->
// -----------C---------D--|-->

Observable<String> source = Observable.create(emitter -> {
    emitter.onNext("A");

    Thread.sleep(500);
    emitter.onNext("B");

    Thread.sleep(200);
    emitter.onNext("C");

    Thread.sleep(800);
    emitter.onNext("D");

    Thread.sleep(600);
    emitter.onNext("E");
    emitter.onComplete();
});

source.subscribeOn(Schedulers.io())
        .throttleLast(1, TimeUnit.SECONDS)
        .blockingSubscribe(
                item -> System.out.println("onNext: " + item),
                Throwable::printStackTrace,
                () -> System.out.println("onComplete"));

// prints:
// onNext: C
// onNext: D
// onComplete
```

throttleLatest

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

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

Emits the next item emitted by a reactive source, then periodically emits the latest item (if any) when the specified timeout elapses between them.

throttleLatest example

```
// Diagram:
// -A----B-C-------D-----E-|-->
// -a------c--1s
//            -----d----1s
```

```
//                      -e-|-->
// -A---------C---------D--|-->

Observable<String> source = Observable.create(emitter -> {
    emitter.onNext("A");

    Thread.sleep(500);
    emitter.onNext("B");

    Thread.sleep(200);
    emitter.onNext("C");

    Thread.sleep(800);
    emitter.onNext("D");

    Thread.sleep(600);
    emitter.onNext("E");
    emitter.onComplete();
});

source.subscribeOn(Schedulers.io())
        .throttleLatest(1, TimeUnit.SECONDS)
        .blockingSubscribe(
                item -> System.out.println("onNext: " + item),
                Throwable::printStackTrace,
                () -> System.out.println("onComplete"));

// prints:
// onNext: A
// onNext: C
// onNext: D
// onComplete
```

throttleWithTimeout

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

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

> *Alias to debounce*

Drops items emitted by a reactive source that are followed by newer items before the given timeout value expires. The timer resets on each emission.

throttleWithTimeout example

```
// Diagram:
// -A--------------B----C-D-------------------E-|---->
//  a---------1s
//                 b---------1s
```

```
//                      c---------1s
//                        d---------1s
//                                              e-|---->
// -----------A---------------------D-----------E-|-->

Observable<String> source = Observable.create(emitter -> {
    emitter.onNext("A");

    Thread.sleep(1_500);
    emitter.onNext("B");

    Thread.sleep(500);
    emitter.onNext("C");

    Thread.sleep(250);
    emitter.onNext("D");

    Thread.sleep(2_000);
    emitter.onNext("E");
    emitter.onComplete();
});

source.subscribeOn(Schedulers.io())
        .throttleWithTimeout(1, TimeUnit.SECONDS)
        .blockingSubscribe(
                item -> System.out.println("onNext: " + item),
                Throwable::printStackTrace,
                () -> System.out.println("onComplete"));

// prints:
// onNext: A
// onNext: D
// onNext: E
// onComplete
```

timeout

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

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

Emits the items from the `Observable` or `Flowable` source, but terminates with a `java.util.concurrent.TimeoutException` if the next item is not emitted within the specified timeout duration starting from the previous item. For `Maybe`, `Single` and `Completable` the specified timeout duration specifies the maximum time to wait for a success or completion event to arrive. If the `Maybe`, `Single` or `Completable` does not complete within the given time a `java.util.concurrent.TimeoutException` will be emitted.

timeout example

```
// Diagram:
// -A-------B---C-----------D-|-->
//  a---------1s
//           b---------1s
//               c---------1s
// -A-------B---C---------X------>

Observable<String> source = Observable.create(emitter -> {
    emitter.onNext("A");

    Thread.sleep(800);
    emitter.onNext("B");

    Thread.sleep(400);
    emitter.onNext("C");

    Thread.sleep(1200);
    emitter.onNext("D");
    emitter.onComplete();
});

source.timeout(1, TimeUnit.SECONDS)
        .subscribe(
                item -> System.out.println("onNext: " + item),
                error -> System.out.println("onError: " + error),
                () -> System.out.println("onComplete will not be printed!"));

// prints:
// onNext: A
// onNext: B
// onNext: C
// onError: java.util.concurrent.TimeoutException: The source did not signal an
event for 1 seconds and has been terminated.
```