

---

# Table of Contents

## Learn RxJS

|                |          |
|----------------|----------|
| Introduction   | 1.1      |
| Operators      | 1.2      |
| Combination    | 1.2.1    |
| combineAll     | 1.2.1.1  |
| combineLatest  | 1.2.1.2  |
| concat         | 1.2.1.3  |
| concatAll      | 1.2.1.4  |
| forkJoin       | 1.2.1.5  |
| merge          | 1.2.1.6  |
| mergeAll       | 1.2.1.7  |
| pairwise       | 1.2.1.8  |
| race           | 1.2.1.9  |
| startWith      | 1.2.1.10 |
| withLatestFrom | 1.2.1.11 |
| zip            | 1.2.1.12 |
| Conditional    | 1.2.2    |
| defaultIfEmpty | 1.2.2.1  |
| every          | 1.2.2.2  |
| Creation       | 1.2.3    |
| create         | 1.2.3.1  |
| empty          | 1.2.3.2  |
| from           | 1.2.3.3  |
| fromEvent      | 1.2.3.4  |
| fromPromise    | 1.2.3.5  |
| interval       | 1.2.3.6  |

---

|                      |          |
|----------------------|----------|
| of                   | 1.2.3.7  |
| range                | 1.2.3.8  |
| throw                | 1.2.3.9  |
| timer                | 1.2.3.10 |
| Error Handling       | 1.2.4    |
| catch                | 1.2.4.1  |
| retry                | 1.2.4.2  |
| retryWhen            | 1.2.4.3  |
| Multicasting         | 1.2.5    |
| publish              | 1.2.5.1  |
| multicast            | 1.2.5.2  |
| share                | 1.2.5.3  |
| Filtering            | 1.2.6    |
| debounce             | 1.2.6.1  |
| debounceTime         | 1.2.6.2  |
| distinctUntilChanged | 1.2.6.3  |
| filter               | 1.2.6.4  |
| first                | 1.2.6.5  |
| ignoreElements       | 1.2.6.6  |
| last                 | 1.2.6.7  |
| sample               | 1.2.6.8  |
| single               | 1.2.6.9  |
| skip                 | 1.2.6.10 |
| skipUntil            | 1.2.6.11 |
| skipWhile            | 1.2.6.12 |
| take                 | 1.2.6.13 |
| takeUntil            | 1.2.6.14 |
| takeWhile            | 1.2.6.15 |
| throttle             | 1.2.6.16 |
| throttleTime         | 1.2.6.17 |

---

---

|                |          |
|----------------|----------|
| Transformation | 1.2.7    |
| buffer         | 1.2.7.1  |
| bufferCount    | 1.2.7.2  |
| bufferTime     | 1.2.7.3  |
| bufferToggle   | 1.2.7.4  |
| bufferWhen     | 1.2.7.5  |
| concatMap      | 1.2.7.6  |
| concatMapTo    | 1.2.7.7  |
| exhaustMap     | 1.2.7.8  |
| expand         | 1.2.7.9  |
| groupBy        | 1.2.7.10 |
| map            | 1.2.7.11 |
| mapTo          | 1.2.7.12 |
| mergeMap       | 1.2.7.13 |
| partition      | 1.2.7.14 |
| pluck          | 1.2.7.15 |
| reduce         | 1.2.7.16 |
| scan           | 1.2.7.17 |
| switchMap      | 1.2.7.18 |
| window         | 1.2.7.19 |
| windowCount    | 1.2.7.20 |
| windowTime     | 1.2.7.21 |
| windowToggle   | 1.2.7.22 |
| windowWhen     | 1.2.7.23 |
| Utility        | 1.2.8    |
| do             | 1.2.8.1  |
| delay          | 1.2.8.2  |
| delayWhen      | 1.2.8.3  |
| dematerialize  | 1.2.8.4  |
| let            | 1.2.8.5  |

---

---

|                                |         |
|--------------------------------|---------|
| toPromise                      | 1.2.8.6 |
| Full Listing                   | 1.2.9   |
| Recipes                        | 1.3     |
| Smart Counter                  | 1.3.1   |
| Concepts                       | 1.4     |
| Understanding Operator Imports | 1.4.1   |

# Learn RxJS

Clear examples, explanations, and resources for RxJS.

## Introduction

[RxJS](#) is one of the hottest libraries in web development today. Offering a powerful, functional approach for dealing with events and with integration points into a growing number of frameworks, libraries, and utilities, the case for learning Rx has never been more appealing. Couple this with the ability to utilize your knowledge across [nearly any language](#), having a solid grasp on reactive programming and what it can offer seems like a no-brainer.

### But...

Learning RxJS and reactive programming is [hard](#). There's the multitude of concepts, large API surface, and fundamental shift in mindset from an [imperative to declarative style](#). This site focuses on making these concepts approachable, the examples clear and easy to explore, and features references throughout to the best RxJS related material on the web. The goal is to supplement the [official docs](#) and pre-existing learning material while offering a new, fresh perspective to clear any hurdles and tackle the pain points. Learning Rx may be difficult but it is certainly worth the effort!

## Content

### Operators

Operators are the horse-power behind observables, providing an elegant, declarative solution to complex asynchronous tasks. This section contains all [RxJS 5 operators](#), included with clear, executable examples in both [JSBin](#) and [JSFiddle](#). Links to additional resources and recipes for each operator are also provided, when applicable.

### Categories

- [Combination](#)
- [Conditional](#)
- [Creation](#)
- [Error Handling](#)
- [Multicasting](#)
- [Filtering](#)
- [Transformation](#)
- [Utility](#)

**OR...**

[Complete listing in alphabetical order](#)

## Concepts

Without a solid base knowledge of how Observables work behind the scenes, it's easy for much of RxJS to feel like 'magic'. This section helps solidify the major concepts needed to feel comfortable with reactive programming and Observables.

[Complete Concept Listing](#)

## Recipes

Recipes for common use-cases and interesting solutions with RxJS.

[Complete Recipe Listing](#)

## Introductory Resources

New to RxJS and reactive programming? In addition to the content found on this site, these excellent articles and videos will help jump start your learning experience!

## Reading

- [RxJS Introduction](#) - Official Docs
- [The Introduction to Reactive Programming You've Been Missing](#) - André Staltz

## Videos

- [Asynchronous Programming: The End of The Loop](#) - Jafar Husain
- [What is RxJS?](#) - Ben Lesh
- [Creating Observable from Scratch](#) - Ben Lesh
- [Introduction to RxJS Marble Testing](#) - Brian Troncone
- [Introduction to Reactive Programming](#) 📺 - André Staltz
- [Reactive Programming using Observables](#) - Jeremy Lund

## Exercises

- [Functional Programming in JavaScript](#) - Jafar Husain

## Tools

- [Rx Marbles - Interactive diagrams of Rx Observables](#) - André Staltz
- [Rx Visualizer - Animated playground for Rx Observables](#) - Misha Moroshko
- [Reactive.how - Animated cards to learn Reactive Programming](#) - Cédric Soulas

Interested in RxJS 4? Check out [Denis Stoyanov's](#) excellent [eBook](#)!

## Translations

- [简体中文](#)

## A Note On References

All references included in this GitBook are resources, both free and paid, that helped me tremendously while learning RxJS. If you come across an article or video that you think should be included, please use the *edit this page* link in the top menu and submit a pull request. Your feedback is appreciated!

# RxJS 5 Operators By Example

A complete list of RxJS 5 operators with clear explanations, relevant resources, and executable examples.

*[Prefer a complete list in alphabetical order?](#)*

## Contents (By Operator Type)

- **Combination**
  - [combineAll](#)
  - [combineLatest](#) ★
  - [concat](#) ★
  - [concatAll](#)
  - [forkJoin](#)
  - [merge](#) ★
  - [mergeAll](#)
  - [race](#)
  - [startWith](#) ★
  - [withLatestFrom](#) ★
  - [zip](#)
- **Conditional**
  - [defaultIfEmpty](#)
  - [every](#)
- **Creation**
  - [create](#)
  - [empty](#)
  - [from](#) ★
  - [fromEvent](#)
  - [fromPromise](#) ★
  - [interval](#)
  - [of](#) ★
  - [range](#)
  - [throw](#)
  - [timer](#)



- Error Handling
  - catch ★
  - retry
  - retryWhen
- Filtering
  - debounce
  - debounceTime ★
  - distinctUntilChanged ★
  - filter ★
  - first
  - ignoreElements
  - last
  - sample
  - single
  - skip
  - skipUntil
  - skipWhile
  - take ★
  - takeUntil ★
  - takeWhile
  - throttle
  - throttleTime
- Multicasting
  - multicast
  - publish
  - share ★
- Transformation
  - buffer
  - bufferCount
  - bufferTime ★
  - bufferToggle
  - bufferWhen
  - concatMap ★
  - concatMapTo
  - expand
  - exhaustMap

- `groupBy`
- `map` ★
- `mapTo`
- `mergeMap` ★
- `partition`
- `pluck`
- `reduce`
- `scan` ★
- `switchMap` ★
- `window`
- `windowCount`
- `windowTime`
- `windowToggle`
- `windowWhen`
- `Utility`
  - `do` ★
  - `delay`
  - `delayWhen`
  - `let`
  - `toPromise`

★ - *commonly used*

## Additional Resources

- [What Are Operators?](#) 📄 - Official Docs
- [What Operators Are](#) 🖥️ 💰 - André Staltz

# Combination Operators

The combination operators allow the joining of information from multiple observables. Order, time, and structure of emitted values is the primary variation among these operators.

## Contents

- [combineAll](#)
- [combineLatest](#) ★
- [concat](#) ★
- [concatAll](#)
- [forkJoin](#)
- [merge](#) ★
- [mergeAll](#)
- [pairwise](#)
- [race](#)
- [startWith](#) ★
- [withLatestFrom](#) ★
- [zip](#)

★ - *commonly used*

## combineAll

**signature:** `combineAll(project: function): Observable`

**When source observable completes use `combineLatest` with collected observables.**

## Examples

( [example tests](#) )

**Example 1: Mapping to inner interval observable**

( [jsBin](#) | [jsFiddle](#) )

```
//emit every 1s, take 2
const source = Rx.Observable.interval(1000).take(2);
//map each emitted value from source to interval observable that
  takes 5 values
const example = source.map(val => Rx.Observable.interval(1000).m
ap(i => `Result (${val}): ${i}`).take(5));
/*
  2 values from source will map to 2 (inner) interval observable
  s that emit every 1s
  combineAll uses combineLatest strategy, emitting the last valu
  e from each
  whenever either observable emits a value
*/
const combined = example.combineAll();
/*
  output:
  ["Result (0): 0", "Result (1): 0"]
  ["Result (0): 1", "Result (1): 0"]
  ["Result (0): 1", "Result (1): 1"]
  ["Result (0): 2", "Result (1): 1"]
  ["Result (0): 2", "Result (1): 2"]
  ["Result (0): 3", "Result (1): 2"]
  ["Result (0): 3", "Result (1): 3"]
  ["Result (0): 4", "Result (1): 3"]
  ["Result (0): 4", "Result (1): 4"]
*/
const subscribe = combined.subscribe(val => console.log(val));
```

## Additional Resources

- [combineAll](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/combineAll.ts>

# combineLatest

**signature:** `combineLatest(observables: ...Observable,  
project: function): Observable`

**When any observable emits a value, emit the latest value from each.**

---

💡 This operator can be used as either a static or instance method!

💡 `combineAll` can be used to apply `combineLatest` to emitted observables when a source completes!

---

## Examples

( [example tests](#) )

**Example 1: Combining observables emitting at 3 intervals**

( [jsBin](#) | [jsFiddle](#) )

```
//timerOne emits first value at 1s, then once every 4s
const timerOne = Rx.Observable.timer(1000, 4000);
//timerTwo emits first value at 2s, then once every 4s
const timerTwo = Rx.Observable.timer(2000, 4000)
//timerThree emits first value at 3s, then once every 4s
const timerThree = Rx.Observable.timer(3000, 4000)

//when one timer emits, emit the latest values from each timer a
s an array
const combined = Rx.Observable
.combineLatest(
  timerOne,
  timerTwo,
  timerThree
);

const subscribe = combined.subscribe(latestValues => {
  //grab latest emitted values for timers one, two, and three
  const [timerValOne, timerValTwo, timerValThree] = latestValu
es;
  /*
    Example:
    timerOne first tick: 'Timer One Latest: 1, Timer Two Latest:
0, Timer Three Latest: 0
    timerTwo first tick: 'Timer One Latest: 1, Timer Two Latest:
1, Timer Three Latest: 0
    timerThree first tick: 'Timer One Latest: 1, Timer Two Lates
t:1, Timer Three Latest: 1
  */
  console.log(
    `Timer One Latest: ${timerValOne},
    Timer Two Latest: ${timerValTwo},
    Timer Three Latest: ${timerValThree}`
  );
});
```

## Example 2: combineLatest with projection function

( [jsBin](#) | [jsFiddle](#) )

```
//timerOne emits first value at 1s, then once every 4s
const timerOne = Rx.Observable.timer(1000, 4000);
//timerTwo emits first value at 2s, then once every 4s
const timerTwo = Rx.Observable.timer(2000, 4000)
//timerThree emits first value at 3s, then once every 4s
const timerThree = Rx.Observable.timer(3000, 4000)

//combineLatest also takes an optional projection function
const combinedProject = Rx.Observable
.combineLatest(
  timerOne,
  timerTwo,
  timerThree,
  (one, two, three) => {
    return `Timer One (Proj) Latest: ${one},
           Timer Two (Proj) Latest: ${two},
           Timer Three (Proj) Latest: ${three}`
  }
);
//log values
const subscribe = combinedProject.subscribe(latestValuesProject =
> console.log(latestValuesProject));
```

### Example 3: Combining events from 2 buttons

( [jsBin](#) | [jsFiddle](#) )



```
// helper function to set HTML
const setHtml = id => val => document.getElementById(id).innerHTML = val;

const addOneClick$ = id => Rx.Observable
  .fromEvent(document.getElementById(id), 'click')
  // map every click to 1
  .mapTo(1)
  .startWith(0)
  // keep a running total
  .scan((acc, curr) => acc + curr)
  // set HTML for appropriate element
  .do(setHtml(`${id}Total`))

const combineTotal$ = Rx.Observable
  .combineLatest(
    addOneClick$('red'),
    addOneClick$('black')
  )
  .map([val1, val2] => val1 + val2)
  .subscribe(setHtml('total'));
```

## HTML

```
<div>
  <button id='red'>Red</button>
  <button id='black'>Black</button>
</div>
<div id="redTotal"></div>
<div id="blackTotal"></div>
<div id="total"></div>
```

## Additional Resources

- [combineLatest](#) 📖 - Official docs
- [Combining streams with combineLatest](#) 🖥️ 💰 - John Linquist
- [Combination operator: combineLatest](#) 🖥️ 💰 - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/combineLatest.ts>

# concat

**signature:** `concat(observables: ...*): Observable`

**Subscribe to observables in order as previous completes, emit values.**

---

💡 You can think of concat like a line at a ATM, the next transaction (subscription) cannot start until the previous completes!

💡 This operator can be used as either a static or instance method!

💡 If throughput order is not a primary concern, try [merge](#) instead!

---

## Examples

( [example tests](#) )

### Example 1: concat 2 basic observables

( [jsBin](#) | [jsFiddle](#) )

```
//emits 1,2,3
const sourceOne = Rx.Observable.of(1,2,3);
//emits 4,5,6
const sourceTwo = Rx.Observable.of(4,5,6);
//emit values from sourceOne, when complete, subscribe to source
Two
const example = sourceOne.concat(sourceTwo);
//output: 1,2,3,4,5,6
const subscribe = example.subscribe(val => console.log('Example:
  Basic concat:', val));
```

### Example 2: concat as static method

( [jsBin](#) | [jsFiddle](#) )

```
//emits 1,2,3
const sourceOne = Rx.Observable.of(1,2,3);
//emits 4,5,6
const sourceTwo = Rx.Observable.of(4,5,6);

//used as static
const example = Rx.Observable.concat(
    sourceOne,
    sourceTwo
);
//output: 1,2,3,4,5,6
const subscribe = example.subscribe(val => console.log('Example:
    static', val));
```

### Example 3: concat with delayed source

( [jsBin](#) | [jsFiddle](#) )

```
//emits 1,2,3
const sourceOne = Rx.Observable.of(1,2,3);
//emits 4,5,6
const sourceTwo = Rx.Observable.of(4,5,6);




//delay 3 seconds then emit
const sourceThree = sourceOne.delay(3000);
//sourceTwo waits on sourceOne to complete before subscribing
const example = sourceThree.concat(sourceTwo);
//output: 1,2,3,4,5,6
const subscribe = example.subscribe(val => console.log('Example:
    Delayed source one:', val));
```

### Example 4: concat with source that does not complete

( [jsBin](#) | [jsFiddle](#) )

```
//when source never completes, the subsequent observables never runs
const source = Rx.Observable
  .concat(
    Rx.Observable.interval(1000),
    Rx.Observable.of('This', 'Never', 'Runs')
  )
//outputs: 1,2,3,4....
const subscribe = source.subscribe(val => console.log('Example:
Source never completes, second observable never runs:', val));
```

## Additional Resources

- [concat](#)  - Official docs
- [Combination operator: concat, startWith](#)   - André Staltz

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/concat.ts>

# concatAll

**signature:** `concatAll(): Observable`

**Collect observables and subscribe to next when previous completes.**

---

⚠ Be wary of [backpressure](#) when the source emits at a faster pace than inner observables complete!

💡 In many cases you can use [concatMap](#) as a single operator instead!

---

## Examples

( [example tests](#) )

### Example 1: concatAll with observable

( [jsBin](#) | [jsFiddle](#) )

```
//emit a value every 2 seconds
const source = Rx.Observable.interval(2000);
const example = source
  //for demonstration, add 10 to and return as observable
  .map(val => Rx.Observable.of(val + 10))
  //merge values from inner observable
  .concatAll();
//output: 'Example with Basic Observable 10', 'Example with Basic Observable 11'...
const subscribe = example.subscribe(val => console.log('Example with Basic Observable:', val));
```

### Example 2: concatAll with promise

[\(jsBin | jsFiddle\)](#)

```
//create and resolve basic promise
const samplePromise = val => new Promise(resolve => resolve(val))
;
//emit a value every 2 seconds
const source = Rx.Observable.interval(2000);

const example = source
  .map(val => samplePromise(val))
  //merge values from resolved promise
  .concatAll();
//output: 'Example with Promise 0', 'Example with Promise 1'...
const subscribe = example.subscribe(val => console.log('Example
with Promise:', val));
```

**Example 3: Delay while inner observables complete**[\(jsBin | jsFiddle\)](#)

```
const obs1 = Rx.Observable.interval(1000).take(5);
const obs2 = Rx.Observable.interval(500).take(2);
const obs3 = Rx.Observable.interval(2000).take(1);
//emit three observables
const source = Rx.Observable.of(obs1, obs2, obs3);
//subscribe to each inner observable in order when previous comp
letes
const example = source.concatAll();
/*
  output: 0,1,2,3,4,0,1,0
  How it works...
  Subscribes to each inner observable and emit values, when comp
lete subscribe to next
  obs1: 0,1,2,3,4 (complete)
  obs2: 0,1 (complete)
  obs3: 0 (complete)
*/

const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [concatAll](#)  - Official docs
  - [Flatten a higher order observable with concatAll in RxJS](#)   - André Staltz
- 



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/concatAll.ts>



# forkJoin

**signature:** `forkJoin(...args, selector : function): Observable`

**When all observables complete, emit the last value from each.**

---

💡 If you want corresponding emissions from multiple observables as they occur, try [zip!](#)

---

## Examples

**Example 1: Making variable number of requests**

( [jsBin](#) | [jsFiddle](#) )

```
const myPromise = val => new Promise(resolve => setTimeout(() =>
  resolve(`Promise Resolved: ${val}`), 5000))

/*
  when all observables complete, give the last
  emitted value from each as an array
*/
const example = Rx.Observable.forkJoin(
  //emit 'Hello' immediately
  Rx.Observable.of('Hello'),
  //emit 'World' after 1 second
  Rx.Observable.of('World').delay(1000),
  //emit 0 after 1 second
  Rx.Observable.interval(1000).take(1),
  //emit 0...1 in 1 second interval
  Rx.Observable.interval(1000).take(2),
  //promise that resolves to 'Promise Resolved' after 5 seconds
  myPromise('RESULT')
);
//output: ["Hello", "World", 0, 1, "Promise Resolved: RESULT"]
const subscribe = example.subscribe(val => console.log(val));

//make 5 requests
const queue = Rx.Observable.of([1,2,3,4,5]);
//emit array of all 5 results
const exampleTwo = queue
  .mergeMap(q => Rx.Observable.forkJoin(...q.map(myPromise)));
/*
  output:
  [
    "Promise Resolved: 1",
    "Promise Resolved: 2",
    "Promise Resolved: 3",
    "Promise Resolved: 4",
    "Promise Resolved: 5"
  ]
*/
const subscribeTwo = exampleTwo.subscribe(val => console.log(val)
);
```

## Additional Resources

- [forkJoin](#)  - Official docs
- 



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/observable/ForkJoinObservable.ts>

## merge

**signature:** `merge(input: Observable): Observable`

**Turn multiple observables into a single observable.**

---

💡 This operator can be used as either a static or instance method!

💡 If order not throughput is a primary concern, try [concat](#) instead!

---

## Examples

**Example 1: merging multiple observables, static method**

( [jsBin](#) | [jsFiddle](#) )

```
//emit every 2.5 seconds
const first = Rx.Observable.interval(2500);
//emit every 2 seconds
const second = Rx.Observable.interval(2000);
//emit every 1.5 seconds
const third = Rx.Observable.interval(1500);
//emit every 1 second
const fourth = Rx.Observable.interval(1000);

//emit outputs from one observable
const example = Rx.Observable.merge(
  first.mapTo('FIRST!'),
  second.mapTo('SECOND!'),
  third.mapTo('THIRD'),
  fourth.mapTo('FOURTH')
);
//output: "FOURTH", "THIRD", "SECOND!", "FOURTH", "FIRST!", "THIRD", "FOURTH"
const subscribe = example.subscribe(val => console.log(val));
```

## Example 2: merge 2 observables, instance method

([jsBin](#) | [jsFiddle](#))

```
//emit every 2.5 seconds
const first = Rx.Observable.interval(2500);
//emit every 1 second
const second = Rx.Observable.interval(1000);
//used as instance method
const example = first.merge(second);
//output: 0,1,0,2....
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [merge](#) 📖 - Official docs
- [Handling multiple streams with merge](#) 🖥️ 💻 - John Linquist
- [Sharing network requests with merge](#) 🖥️ 💻 - André Staltz
- [Combination operator: merge](#) 🖥️ 💻 - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/merge.ts>

# mergeAll

**signature:** `mergeAll(concurrent: number): Observable`

**Collect and subscribe to all observables.**

---

💡 In many cases you can use [mergeMap](#) as a single operator instead!

---

## Examples

( [example tests](#) )

**Example 1: mergeAll with promises**

( [jsBin](#) | [jsFiddle](#) )

```
const myPromise = val => new Promise(resolve => setTimeout(() =>
  resolve(`Result: ${val}`), 2000))
//emit 1,2,3
const source = Rx.Observable.of(1,2,3);

const example = source
  //map each value to promise
  .map(val => myPromise(val))
  //emit result from source
  .mergeAll();

/*
  output:
  "Result: 1"
  "Result: 2"
  "Result: 3"
*/
const subscribe = example.subscribe(val => console.log(val));
```

## Example 2: mergeAll with *concurrent* parameter

( [jsFiddle](#) )



```
console.clear();

const interval = Rx.Observable.interval(500).take(5);

/*
  interval is emitting a value every 0.5s. This value is then being mapped to interval that
  is delayed for 1.0s. The mergeAll operator takes an optional argument that determines how
  many inner observables to subscribe to at a time. The rest of the observables are stored
  in a backlog waiting to be subscribe.
*/
const example = interval
  .map(val => interval.delay(1000).take(3))
  .mergeAll(2)
  .subscribe(val => console.log(val));
/*
  The subscription is completed once the operator emits all values.
*/
```

## Additional Resources

- [mergeAll](#) 📖 - Official docs
- [Flatten a higher order observable with mergeAll in RxJS](#) 📖 💡 - André Staltz

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/mergeAll.ts>

# pairwise

**signature:** `pairwise(): Observable<Array>`

**Emit the previous and current values as an array.**

## Examples

**Example 1:**

([jsBin](#) | [jsFiddle](#))

```
var interval = Rx.Observable.interval(1000);

//Returns: [0,1], [1,2], [2,3], [3,4], [4,5]
interval.pairwise()
  .take(5)
  .subscribe(console.log);
```

## Additional Resources

- [pairwise](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/pairwise.ts>

## race

**signature:** `race(): Observable`

**The observable to emit first is used.**

## Examples

### Example 1: race with 4 observables

( [jsBin](#) | [jsFiddle](#) )

```
//take the first observable to emit
const example = Rx.Observable.race(
  //emit every 1.5s
  Rx.Observable.interval(1500),
  //emit every 1s
  Rx.Observable.interval(1000).mapTo('1s won!'),
  //emit every 2s
  Rx.Observable.interval(2000),
  //emit every 2.5s
  Rx.Observable.interval(2500)
);
//output: "1s won!"..."1s won!"...etc
const subscribe = example.subscribe(val => console.log(val));
```

### Example 2: race with an error


( [jsFiddle](#) )

```
console.clear();

//Throws an error and ignore the rest of the observables.
const first = Rx.Observable.of('first').delay(100).map(() => {throw 'error'});
const second = Rx.Observable.of('second').delay(200);
const third = Rx.Observable.of('third').delay(300);

const race = Rx.Observable.race(first, second, third)
    .subscribe(val => console.log(val));
```

## Additional Resources

- [race](#)  - Official docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/race.ts>

# startWith

**signature:** `startWith(an: Values): Observable`

**Emit given value first.**

---

💡 A [BehaviorSubject](#) can also start with an initial value!

---

## Examples

( [example tests](#) )

**Example 1: startWith on number sequence**

( [jsBin](#) | [jsFiddle](#) )

```
//emit (1,2,3)
const source = Rx.Observable.of(1,2,3);
//start with 0
const example = source.startWith(0);
//output: 0,1,2,3
const subscribe = example.subscribe(val => console.log(val));
```

**Example 2: startWith for initial scan value**

( [jsBin](#) | [jsFiddle](#) )

```
//emit ('World!', 'Goodbye', 'World!')
const source = Rx.Observable.of('World!', 'Goodbye', 'World!');
//start with 'Hello', concat current string to previous
const example = source
  .startWith('Hello')
  .scan((acc, curr) => `${acc} ${curr}`);
/*
  output:
  "Hello"
  "Hello World!"
  "Hello World! Goodbye"
  "Hello World! Goodbye World!"
*/
const subscribe = example.subscribe(val => console.log(val));
```

### Example 3: startWith multiple values

( [jsBin](#) | [jsFiddle](#) )

```
//emit values in sequence every 1s
const source = Rx.Observable.interval(1000);
//start with -3, -2, -1
const example = source.startWith(-3, -2, -1);
//output: -3, -2, -1, 0, 1, 2....
const subscribe = example.subscribe(val => console.log(val));
```

## Related Recipes

- [Smart Counter](#)

## Additional Resources

- [startWith](#) 📖 - Official docs
- [Displaying initial data with startWith](#) 🖥️ 💰 - John Linquist
- [Clear data while loading with startWith](#) 🖥️ 💰 - André Staltz
- [Combination operator: concat, startWith](#) 🖥️ 💰 - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/startWith.ts>

## withLatestFrom

**signature:** `withLatestFrom(other: Observable, project: Function): Observable`

**Also provide the last value from another observable.**

---

💡 If you want the last emission any time a variable number of observables emits, try [combineLatest!](#)

---

## Examples

**Example 1: Latest value from quicker second source**

([jsBin](#) | [jsFiddle](#))

```
//emit every 5s
const source = Rx.Observable.interval(5000);
//emit every 1s
const secondSource = Rx.Observable.interval(1000);
const example = source
  .withLatestFrom(secondSource)
  .map(([first, second]) => {
    return `First Source (5s): ${first} Second Source (1s): ${second}`;
  });
/*
  "First Source (5s): 0 Second Source (1s): 4"
  "First Source (5s): 1 Second Source (1s): 9"
  "First Source (5s): 2 Second Source (1s): 14"
  ...
*/
const subscribe = example.subscribe(val => console.log(val));
```






## Example 2: Slower second source

([jsBin](#) | [jsFiddle](#))

```
//emit every 5s
const source = Rx.Observable.interval(5000);
//emit every 1s
const secondSource = Rx.Observable.interval(1000);
//withLatestFrom slower than source
const example = secondSource
  //both sources must emit at least 1 value (5s) before emitting
  .withLatestFrom(source)
  .map(([first, second]) => {
    return `Source (1s): ${first} Latest From (5s): ${second}`;
  });
/*
  "Source (1s): 4 Latest From (5s): 0"
  "Source (1s): 5 Latest From (5s): 0"
  "Source (1s): 6 Latest From (5s): 0"
  ...
*/
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [withLatestFrom](#)  - Official docs
- [Combination operator: withLatestFrom](#)   - André Staltz

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/withLatestFrom.ts>

# zip

**signature:** `zip(observables: *): Observable`

## Description

**TL;DR:** After all observables emit, emit values as an array

The **zip** operator will subscribe to all inner observables, waiting for each to emit a value. Once this occurs, all values with the corresponding index will be emitted. This will continue until at least one inner observable completes.

---

💡 Combined with [interval](#) or [timer](#), zip can be used to time output from another source!

---

## Examples

**Example 1: zip multiple observables emitting at alternate intervals**

( [jsBin](#) | [jsFiddle](#) )

```
const sourceOne = Rx.Observable.of('Hello');
const sourceTwo = Rx.Observable.of('World!');
const sourceThree = Rx.Observable.of('Goodbye');
const sourceFour = Rx.Observable.of('World!');
//wait until all observables have emitted a value then emit all
as an array
const example = Rx.Observable
  .zip(
    sourceOne,
    sourceTwo.delay(1000),
    sourceThree.delay(2000),
    sourceFour.delay(3000)
  );
//output: ["Hello", "World!", "Goodbye", "World!"]
const subscribe = example.subscribe(val => console.log(val));
```

## Example 2: zip when 1 observable completes

( [jsBin](#) | [jsFiddle](#) )

```
//emit every 1s
const interval = Rx.Observable.interval(1000);
//when one observable completes no more values will be emitted
const example = Rx.Observable
  .zip(
    interval,
    interval.take(2)
  );
//output: [0,0]...[1,1]
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [zip](#) 📖 - Official docs
- [Combination operator: zip](#) 🖥️ 💰 - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/zip.ts>

# Conditional Operators

For use-cases that depend on a specific condition to be met, these operators do the trick.

## Contents

- [defaultIfEmpty](#)
- [every](#)

# defaultIfEmpty

**signature:** `defaultIfEmpty(defaultValue: any): Observable`

**Emit given value if nothing is emitted before completion.**

## Examples

### Example 1: Default for empty value

( [jsBin](#) | [jsFiddle](#) )

```
const empty = Rx.Observable.of();
//emit 'Observable.of() Empty!' when empty, else any values from
//source
const exampleOne = empty.defaultIfEmpty('Observable.of() Empty!');
;
//output: 'Observable.of() Empty!'
const subscribe = exampleOne.subscribe(val => console.log(val));
```

### Example 2: Default for Observable.empty

( [jsBin](#) | [jsFiddle](#) )

```
//empty observable
const empty = Rx.Observable.empty();
//emit 'Observable.empty()!' when empty, else any values from so
//urce
const example = empty.defaultIfEmpty('Observable.empty()!');
//output: 'Observable.empty()!'
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [defaultIfEmpty](#)  - Official docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/defaultIfEmpty.ts>

## every

**signature:** `every(predicate: function, thisArg: any): Observable`

**If all values pass predicate before completion emit true, else false.**

## Examples

### Example 1: Some values false

([jsBin](#) | [jsFiddle](#))

```
//emit 5 values
const source = Rx.Observable.of(1,2,3,4,5);
const example = source
  //is every value even?
  .every(val => val % 2 === 0)
//output: false
const subscribe = example.subscribe(val => console.log(val));
```


### Example 2: All values true

([jsBin](#) | [jsFiddle](#))

```
//emit 5 values
const allEvens = Rx.Observable.of(2,4,6,8,10);
const example = allEvens
  //is every value even?
  .every(val => val % 2 === 0);
//output: true
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources



- [every](#)  - Official docs
- 

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/every.ts>

# Creation Operators

These operators allow the creation of an observable from nearly anything. From generic to specific use-cases you are free, and encouraged, to turn [everything into a stream](#).

## Contents

- [create](#)
- [empty](#)
- [from](#) ★
- [fromEvent](#)
- [fromPromise](#) ★
- [interval](#)
- [of](#) ★
- [range](#)
- [throw](#)
- [timer](#)

★ - *commonly used*

## Additional Resources

- [Creating Observables From Scratch](#) 📄 💻 - André Staltz

# create

**signature:** `create(subscribe: function)`

**Create an observable with given subscription function.**

## Examples

**Example 1: Observable that emits multiple values**

( [jsBin](#) | [jsFiddle](#) )

```
/*
  Create an observable that emits 'Hello' and 'World' on
  subscription.
*/
const hello = Rx.Observable.create(function(observer) {
  observer.next('Hello');
  observer.next('World');
});

//output: 'Hello'...'World'
const subscribe = hello.subscribe(val => console.log(val));
```

**Example 2: Observable that emits even numbers on timer**

( [jsBin](#) | [jsFiddle](#) )

```
/*
  Increment value every 1s, emit even numbers.
*/
const evenNumbers = Rx.Observable.create(function(observer) {
  let value = 0;
  const interval = setInterval(() => {
    if(value % 2 === 0){
      observer.next(value);
    }
    value++;
  }, 1000);

  return () => clearInterval(interval);
});
//output: 0...2...4...6...8
const subscribe = evenNumbers.subscribe(val => console.log(val));

//unsubscribe after 10 seconds
setTimeout(() => {
  subscribe.unsubscribe();
}, 10000);
```

## Additional Resources

- [create](#) 📄 - Official docs
- [Creation operators: Create\(\)](#) 📄 📄 - André Staltz
- [Using Observable.create for fine-grained control](#) 📄 📄 - Shane Osbourne

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/observable/GenerateObservable.ts>

# empty

**signature:** `empty(scheduler: Scheduler): Observable`

**Observable that immediately completes.**

## Examples

**Example 1: empty immediately completes**




( [jsBin](#) | [jsFiddle](#) )

```
//Create observable that immediately completes
const example = Rx.Observable.empty();
//output: 'Complete!'
const subscribe = example.subscribe({
  next: () => console.log('Next'),
  complete: () => console.log('Complete!')
});
```

## Follow the Source Code

*Coming soon...*

## Additional Resources

- [empty](#)  - Official docs
- [Creation operators: empty, never, and throw](#)   - André Staltz

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/observable/EmptyObservable.ts>



# from

**signature:** `from(ish: ObservableInput, mapFn: function, thisArg: any, scheduler: Scheduler): Observable`

## Turn an array, promise, or iterable into an observable.

---

💡 For arrays and iterables, all contained values will be emitted as a sequence!

💡 This operator can also be used to emit a string as a sequence of characters!

---

## Examples

### Example 1: Observable from array


( [jsBin](#) | [jsFiddle](#) )

```
//emit array as a sequence of values
const arraySource = Rx.Observable.from([1,2,3,4,5]);
//output: 1,2,3,4,5
const subscribe = arraySource.subscribe(val => console.log(val));
```

### Example 2: Observable from promise

( [jsBin](#) | [jsFiddle](#) )

```
//emit result of promise
const promiseSource = Rx.Observable.from(new Promise(resolve =>
  resolve('Hello World!')));
//output: 'Hello World'
const subscribe = promiseSource.subscribe(val => console.log(val)
);
```



### Example 3: Observable from collection

( [jsBin](#) | [jsFiddle](#) )

```
//works on js collections
const map = new Map();
map.set(1, 'Hi');
map.set(2, 'Bye');


const mapSource = Rx.Observable.from(map);
//output: [1, 'Hi'], [2, 'Bye']
const subscribe = mapSource.subscribe(val => console.log(val));
```

### Example 4: Observable from string

( [jsBin](#) | [jsFiddle](#) )

```
//emit string as a sequence
const source = Rx.Observable.from('Hello World');
//output: 'H','e','l','l','o',' ','W','o','r','l','d'
const subscribe = source.subscribe(val => console.log(val));
```

## Additional Resources

- [from](#)  - Official docs
- [Creation operators: from, fromArray, fromPromise](#)   - André Staltz



from

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/observable/FromObservable.ts>

# fromEvent

**signature:** `fromEvent(target: EventTargetLike, eventName: string, selector: function): Observable`

**Turn event into observable sequence.**

## Examples

**Example 1: Observable from mouse clicks**

( [jsBin](#) | [jsFiddle](#) )

```
//create observable that emits click events
const source = Rx.Observable.fromEvent(document, 'click');
//map to string with given event timestamp
const example = source.map(event => `Event time: ${event.timeSta
mp}`)
//output (example): 'Event time: 7276.3900000000001'
const subscribe = example.subscribe(val => console.log(val));
```

## Related Recipes

- [Smart Counter](#)

## Additional Resources

- [fromEvent](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/observable/FromEventObservable.ts>



# fromPromise

**signature:** `fromPromise(promise: Promise, scheduler: Scheduler): Observable`

**Create observable from promise, emitting result.**

---

💡 Flattening operators can generally accept promises without wrapping!

💡 You could also use [Observable.from](#) for the same result!

---

## Examples

**Example 1: Converting promise to observable and catching errors**

( [jsBin](#) | [jsFiddle](#) )

```
//example promise that will resolve or reject based on input
const myPromise = (willReject) => {
  return new Promise((resolve, reject) => {
    if(willReject){
      reject('Rejected!');
    }
    resolve('Resolved!');
  })
}
//emit true, then false
const source = Rx.Observable.of(true, false);
const example = source
  .mergeMap(val => Rx.Observable
    //turn promise into observable
    .fromPromise(myPromise(val))
    //catch and gracefully handle rejections
    .catch(error => Rx.Observable.of(`Error: ${error}`)))
  )
//output: 'Error: Rejected!', 'Resolved!'
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [fromPromise](#) 📖 - Official docs
- [Creation operators: from, fromArray, fromPromise](#) 🖨️ 💡 - André Staltz
- [fromPromise - Guide](#)

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/observable/PromiseObservable.ts>

# interval

**signature:** `interval(period: number, scheduler: Scheduler): Observable`

**Emit numbers in sequence based on provided timeframe.**

## Examples

**Example 1: Emit sequence of values at 1 second interval**

( [jsBin](#) | [jsFiddle](#) )

```
//emit value in sequence every 1 second
const source = Rx.Observable.interval(1000);
//output: 0,1,2,3,4,5....
const subscribe = source.subscribe(val => console.log(val));
```

## Additional Resources

- [interval](#)  - Official docs
- [Creation operators: interval and timer](#)   - André Staltz

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/observable/IntervalObservable.ts>

## of

**signature:** `of(...values, scheduler: Scheduler): Observable`

**Emit variable amount of values in a sequence.**

## Examples

### Example 1: Emitting a sequence of numbers

( [jsBin](#) | [jsFiddle](#) )

```
//emits any number of provided values in sequence
const source = Rx.Observable.of(1,2,3,4,5);
//output: 1,2,3,4,5
const subscribe = source.subscribe(val => console.log(val));
```

### Example 2: Emitting an object, array, and function

( [jsBin](#) | [jsFiddle](#) )

```
//emits values of any type
const source = Rx.Observable.of({name: 'Brian'}, [1,2,3], function hello(){ return 'Hello'});
//output: {name: 'Brian'}, [1,2,3], function hello() { return 'Hello' }
const subscribe = source.subscribe(val => console.log(val));
```

## Additional Resources

- [of](#) 📖 - Official docs
- [Creation operators: of](#) 🖥️ 💡 - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/observable/ArrayObservable.ts>



# range

**signature:** `range(start: number, count: number, scheduler: Scheduler): Observable`

**Emit numbers in provided range in sequence.**


## Examples

**Example 1: Emit range 1-10**

([jsBin](#) | [jsFiddle](#) )

```
//emit 1-10 in sequence
const source = Rx.Observable.range(1,10);
//output: 1,2,3,4,5,6,7,8,9,10
const example = source.subscribe(val => console.log(val));
```

## Additional Resources

- [range](#)  - Official docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/observable/RangeObservable.ts>

# throw

**signature:** `throw(error: any, scheduler: Scheduler): Observable`

## Emit error on subscription.

## Examples

### Example 1: Throw error on subscription

([jsBin](#) | [jsFiddle](#))

```
//emits an error with specified value on subscription
const source = Rx.Observable.throw('This is an error!');
//output: 'Error: This is an error!'
const subscribe = source.subscribe({
  next: val => console.log(val),
  complete: () => console.log('Complete!'),
  error: val => console.log(`Error: ${val}`)
});
```

## Additional Resources

- [throw](#) 📄 - Official docs
- [Creation operators: empty, never, and throw](#) 🖥️ 📦 - André Staltz

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/observable/ErrorObservable.ts>

# timer

**signature:** `timer(initialDelay: number | Date, period: number, scheduler: Scheduler): Observable`

**After given duration, emit numbers in sequence every specified duration.**

## Examples

**Example 1: timer emits 1 value then completes**

([jsBin](#) | [jsFiddle](#))




```
//emit 0 after 1 second then complete, since no second argument
is supplied
const source = Rx.Observable.timer(1000);
//output: 0
const subscribe = source.subscribe(val => console.log(val));
```

**Example 2: timer emits after 1 second, then every 2 seconds**

([jsBin](#) | [jsFiddle](#))

```
/*
    timer takes a second argument, how often to emit subsequent va
lues
    in this case we will emit first value after 1 second and subse
quent
    values every 2 seconds after
*/
const source = Rx.Observable.timer(1000, 2000);
//output: 0,1,2,3,4,5.....
const subscribe = source.subscribe(val => console.log(val));
```

## Additional Resources

- [timer](#)  - Official docs
  - [Creation operators: interval and timer](#)   - André Staltz
- 

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/observable/TimerObservable.ts>

# Error Handling Operators

Errors are an unfortunate side-effect of development. These operators provide effective ways to gracefully handle errors and retry logic, should they occur.

## Contents

- `catch` ★
- `retry`
- `retryWhen`

★ - *commonly used*

# catch

**signature:** `catch(project : function): Observable`

## Gracefully handle errors in an observable sequence.

---

⚠ Remember to return an observable from the catch function!

---

## Examples

( [example tests](#) )

### Example 1: Catching error from observable

( [jsBin](#) | [jsFiddle](#) )

```
//emit error
const source = Rx.Observable.throw('This is an error!');
//gracefully handle error, returning observable with error message
const example = source.catch(val => Rx.Observable.of(`I caught: ${val}`));
//output: 'I caught: This is an error'
const subscribe = example.subscribe(val => console.log(val));
```

### Example 2: Catching rejected promise

( [jsBin](#) | [jsFiddle](#) )

```
//create promise that immediately rejects
const myBadPromise = () => new Promise((resolve, reject) => reject('Rejected!'));
//emit single value after 1 second
const source = Rx.Observable.timer(1000);
//catch rejected promise, returning observable containing error message
const example = source.flatMap(() => Rx.Observable
    .fromPromise(myBadPromise())
    .catch(error => Rx.Observable.of(`Bad Promise: ${error}`)));
//output: 'Bad Promise: Rejected'
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [Error handling operator: catch](#) 📖 📄 - André Staltz

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/catch.ts>

## retry

**signature:** `retry(number: number): Observable`

**Retry an observable sequence a specific number of times should an error occur.**

## Examples

**Example 1: Retry 2 times on error**

( [jsBin](#) | [jsFiddle](#) )



```
//emit value every 1s
const source = Rx.Observable.interval(1000);
const example = source
    .flatMap(val => {
        //throw error for demonstration
        if(val > 5){
            return Rx.Observable.throw('Error!');
        }
        return Rx.Observable.of(val);
    })
    .retry(2);
/*
output:
0..1..2..3..4..5..
0..1..2..3..4..5..
0..1..2..3..4..5..
"Error!: Retried 2 times then quit!"
*/
const subscribe = example
    .subscribe({
        next: val => console.log(val),
        error: val => console.log(`${val}: Retried 2 times then quit!`)
    });
```

## Additional Resources

- [retry](#) 📖 - Official docs
- [Error handling operator: retry and retryWhen](#) 🖥️ 💻 - André Staltz

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/retry.ts>

## retryWhen

**signature:** `retryWhen(receives: (errors: Observable) => Observable, the: scheduler): Observable`

**Retry an observable sequence on error based on custom criteria.**

### Examples

**Example 1: Trigger retry after specified duration**

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 1s
const source = Rx.Observable.interval(1000);
const example = source
    .map(val => {
        if(val > 5){
            //error will be picked up by retryWhen
            throw val;
        }
        return val;
    })
    .retryWhen(errors => errors
        //log error message
        .do(val => console.log(`Value ${val} was too high
!`))

        //restart in 5 seconds
        .delayWhen(val => Rx.Observable.timer(val * 1000))

    );

/*
output:
0
1
2
3
4
5
"Value 6 was too high!"
--Wait 5 seconds then repeat
*/
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [retryWhen](#) 📄 - Official docs
- [Error handling operator: retry and retryWhen](#) 🖥️ 💡 - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/retryWhen.ts>

# Multicasting Operators

In RxJS observables are cold, or unicast by default. These operators can make an observable hot, or multicast, allowing side-effects to be share among multiple subscribers.

## Contents

- [publish](#)
- [multicast](#)
- [share](#) ★

★ - *commonly used*

## Additional Resources

- [Hot vs Cold Observables](#) 📄 - Ben Lesh
- [Unicast v Multicast](#) 📄 - GitHub Discussion
- [Demystifying Hot and Cold Observables](#) 📺 - André Staltz

## publish

**signature:** `publish() : ConnectableObservable`

**Share source and make hot by calling connect.**

## Examples

**Example 1: Connect observable after subscribers**

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 1 second
const source = Rx.Observable.interval(1000);
const example = source
  //side effects will be executed once
  .do(() => console.log('Do Something!'))
  //do nothing until connect() is called
  .publish();

/*
  source will not emit values until connect() is called
  output: (after 5s)
  "Do Something!"
  "Subscriber One: 0"
  "Subscriber Two: 0"
  "Do Something!"
  "Subscriber One: 1"
  "Subscriber Two: 1"
*/
const subscribe = example.subscribe(val => console.log(`Subscriber One: ${val}`));
const subscribeTwo = example.subscribe(val => console.log(`Subscriber Two: ${val}`));

//call connect after 5 seconds, causing source to begin emitting items
setTimeout(() => {
  example.connect();
}, 5000)
```

## Additional Resources

- [publish](#)  - Official docs

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/publish.ts>

# multicast

**signature:** `multicast(selector: Function): Observable`

**Share source utilizing the provided Subject.**

## Examples

### Example 1: multicast with standard Subject

( [jsBin](#) | [jsFiddle](#) )

```
//emit every 2 seconds, take 5
const source = Rx.Observable.interval(2000).take(5);

const example = source
  //since we are multicasting below, side effects will be executed once
  .do(() => console.log('Side Effect #1'))
  .mapTo('Result!')

//subscribe subject to source upon connect()
const multi = example.multicast(() => new Rx.Subject());
/*
  subscribers will share source
  output:
  "Side Effect #1"
  "Result!"
  "Result!"
  ...
*/
const subscriberOne = multi.subscribe(val => console.log(val));
const subscriberTwo = multi.subscribe(val => console.log(val));
//subscribe subject to source
multi.connect();
```

### Example 2: multicast with ReplaySubject



([jsBin](#) | [jsFiddle](#))

```
//emit every 2 seconds, take 5
const source = Rx.Observable.interval(2000).take(5);

//example with ReplaySubject
const example = source
  //since we are multicasting below, side effects will be executed once
  .do(() => console.log('Side Effect #2'))
  .mapTo('Result Two!')
//can use any type of subject
const multi = example.multicast(() => new Rx.ReplaySubject(5));
//subscribe subject to source
multi.connect();

setTimeout(() => {
  /*
    subscriber will receive all previous values on subscription
    because
    of ReplaySubject
    */
  const subscriber = multi
    .subscribe(val => console.group(val));
}, 5000);
```

## Additional Resources

- [multicast](#)  - Official docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/multicast.ts>

# share

**signature:** `share(): Observable`

**Share source among multiple subscribers.**

---

💡 share is like [multicast](#) with a Subject and refCount!

---

## Examples

**Example 1: Multiple subscribers sharing source**

( [jsBin](#) | [jsFiddle](#) )

```
//emit value in 1s
const source = Rx.Observable.timer(1000);
//log side effect, emit result
const example = source
  .do(() => console.log('***SIDE EFFECT***'))
  .mapTo('***RESULT***');
/*
  ***NOT SHARED, SIDE EFFECT WILL BE EXECUTED TWICE***
  output:
  "***SIDE EFFECT***"
  "***RESULT***"
  "***SIDE EFFECT***"
  "***RESULT***"
*/
const subscribe = example.subscribe(val => console.log(val));
const subscribeTwo = example.subscribe(val => console.log(val));

//share observable among subscribers
const sharedExample = example.share();
/*
  ***SHARED, SIDE EFFECT EXECUTED ONCE***
  output:
  "***SIDE EFFECT***"
  "***RESULT***"
  "***RESULT***"
*/
const subscribeThree = sharedExample.subscribe(val => console.log(val));
const subscribeFour = sharedExample.subscribe(val => console.log(val));
```

## Additional Resources

- [share](#) 📖 - Official docs
- [Sharing streams with share](#) 📺 📄 - John Linquist



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/share.ts>

# Filtering Operators

In a [push based approach](#), picking and choosing how and when to accept items is important. These operators provide techniques for accepting values from an observable source and dealing with [backpressure](#).

## Contents

- [debounce](#)
- [debounceTime](#) ★
- [distinctUntilChanged](#) ★
- [filter](#) ★
- [first](#)
- [ignoreElements](#)
- [last](#)
- [sample](#)
- [single](#)
- [skip](#)
- [skipUntil](#)
- [skipWhile](#)
- [take](#) ★
- [takeUntil](#) ★
- [takeWhile](#)
- [throttle](#)
- [throttleTime](#)

★ - *commonly used*

# debounce

**signature:** `debounce(durationSelector: function): Observable`

**Discard emitted values that take less than the specified time, based on selector function, between output.**

---

💡 Though not as widely used as `debounceTime`, **debounce** is important when the debounce rate is variable!

---

## Examples

### Example 1: Debounce on timer

( [jsBin](#) | [jsFiddle](#) )

```
//emit four strings
const example = Rx.Observable.of('WAIT', 'ONE', 'SECOND', 'Last will display');
/*
    Only emit values after a second has passed between the last
    emission,
    throw away all other values
*/
const debouncedExample = example.debounce(() => Rx.Observable.timer(1000));
/*
    In this example, all values but the last will be omitted
    output: 'Last will display'
*/
const subscribe = debouncedExample.subscribe(val => console.log(val));
```

## Example 2: Debounce at increasing interval

([jsBin](#) | [jsFiddle](#))

```
//emit value every 1 second, ex. 0...1...2
const interval = Rx.Observable.interval(1000);
//raise the debounce time by 200ms each second
const debouncedInterval = interval.debounce(val => Rx.Observable.timer(val * 200))
/*
  After 5 seconds, debounce time will be greater than interval time,
  all future values will be thrown away
  output: 0...1...2...3...4.....(debounce time over 1s, no values emitted)
*/
const subscribe = debouncedInterval.subscribe(val => console.log(`Example Two: ${val}`));
```

## Additional Resources

- [debounce](#) 📄 - Official docs
- [Transformation operator: debounce and debounceTime](#) 📄 📄 - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/debounce.ts>

# debounceTime

**signature:** `debounceTime(dueTime: number, scheduler: Scheduler): Observable`

## Discard emitted values that take less than the specified time between output

---

💡 This operator is popular in scenarios such as type-ahead where the rate of user input must be controlled!

---

## Examples

### Example 1: Debouncing based on time between input

([jsBin](#) | [jsFiddle](#))

```
const input = document.getElementById('example');

//for every keyup, map to current input value
const example = Rx.Observable
  .fromEvent(input, 'keyup')
  .map(i => i.currentTarget.value);

//wait .5s between keyups to emit current value
//throw away all other values
const debouncedInput = example.debounceTime(500);

//log values
const subscribe = debouncedInput.subscribe(val => {
  console.log(`Debounced Input: ${val}`);
});
```



## Additional Resources

- [debounceTime](#)  - Official docs
  - [Transformation operator: debounce and debounceTime](#)   - André Staltz
- 

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/debounceTime.ts>

# distinctUntilChanged

**signature:** `distinctUntilChanged(compare: function): Observable`

**Only emit when the current value is different than the last.**

---

💡 `distinctUntilChanged` uses `===` comparison by default, object references must match!

---

## Examples

### Example 1: `distinctUntilChanged` with basic values

( [jsBin](#) | [jsFiddle](#) )

```
//only output distinct values, based on the last emitted value
const myArrayWithDuplicatesInARow = Rx.Observable
  .from([1,1,2,2,3,1,2,3]);

const distinctSub = myArrayWithDuplicatesInARow
  .distinctUntilChanged()
  //output: 1,2,3,1,2,3
  .subscribe(val => console.log('DISTINCT SUB:', val));

const nonDistinctSub = myArrayWithDuplicatesInARow
  //output: 1,1,2,2,3,1,2,3
  .subscribe(val => console.log('NON DISTINCT SUB:', val));
```

### Example 2: `distinctUntilChanged` with objects

( [jsBin](#) | [jsFiddle](#) )

```
const sampleObject = {name: 'Test'};
//Objects must be same reference
const myArrayWithDuplicateObjects = Rx.Observable.from([sampleObject, sampleObject, sampleObject]);
//only out distinct objects, based on last emitted value
const nonDistinctObjects = myArrayWithDuplicateObjects
    .distinctUntilChanged()
    //output: 'DISTINCT OBJECTS: {name: 'Test'}
    .subscribe(val => console.log('DISTINCT OBJECTS:', val));
```

## Additional Resources

- [distinctUntilChanged](#) 📄 - Official docs
- [Filtering operator: distinct and distinctUntilChanged](#) 🖥️ 📄 - André Staltz

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/distinctUntilChanged.ts>

# filter

**signature:** `filter(select: Function, thisArg: any): Observable`

**Emit values that pass the provided condition.**

---

💡 If you would like to complete an observable when a condition fails, check out [takeWhile!](#)

---

## Examples

**Example 1: filter for even numbers**

( [jsBin](#) | [jsFiddle](#) )

```
//emit (1,2,3,4,5)
const source = Rx.Observable.from([1,2,3,4,5]);
//filter out non-even numbers
const example = source.filter(num => num % 2 === 0);
//output: "Even number: 2", "Even number: 4"
const subscribe = example.subscribe(val => console.log(`Even number: ${val}`));
```

**Example 2: filter objects based on property**

( [jsBin](#) | [jsFiddle](#) )

```
//emit ({name: 'Joe', age: 31}, {name: 'Bob', age:25})
const source = Rx.Observable.from([{name: 'Joe', age: 31}, {name:
  'Bob', age:25}]);
//filter out people with age under 30
const example = source.filter(person => person.age >= 30);
//output: "Over 30: Joe"
const subscribe = example.subscribe(val => console.log(`Over 30:
${val.name}`));
```

### Example 3: filter for number greater than specified value

( [jsBin](#) | [jsFiddle](#) )

```
//emit every second
const source = Rx.Observable.interval(1000);
//filter out all values until interval is greater than 5
const example = source.filter(num => num > 5);
/*
  "Number greater than 5: 6"
  "Number greater than 5: 7"
  "Number greater than 5: 8"
  "Number greater than 5: 9"
*/
const subscribe = example.subscribe(val => console.log(`Number g
reater than 5: ${val}`));
```

## Additional Resources

- [filter](#) 📄 - Official docs
- [Adding conditional logic with filter](#) 🖥️ 💰 📄 - John Linquist
- [Filtering operator: filter](#) 🖥️ 💰 📄 - André Staltz

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/filter.ts>



# first

**signature:** `first(predicate: function, select: function)`

**Emit the first value or first to pass provided expression.**

---

💡 The counterpart to first is **last**!

---

## Examples

( [example tests](#) )

### Example 1: First value from sequence

( [jsBin](#) | [jsFiddle](#) )

```
const source = Rx.Observable.from([1,2,3,4,5]);
//no arguments, emit first value
const example = source.first();
//output: "First value: 1"
const subscribe = example.subscribe(val => console.log(`First value: ${val}`));
```

### Example 2: First value to pass predicate

( [jsBin](#) | [jsFiddle](#) )

```
const source = Rx.Observable.from([1,2,3,4,5]);
//emit first item to pass test
const example = source.first(num => num === 5);
//output: "First to pass test: 5"
const subscribe = example.subscribe(val => console.log(`First to
pass test: ${val}`));
```

### Example 3: Using optional projection function

([jsBin](#) | [jsFiddle](#))

```
const source = Rx.Observable.from([1,2,3,4,5]);
//using optional projection function
const example = source.first(num => num % 2 === 0,
                             (result, index) => `First ev
en: ${result} is at index: ${index}`);
//output: "First even: 2 at index: 1"
const subscribe = example.subscribe(val => console.log(val));
```

### Example 4: Utilizing default value

([jsBin](#) | [jsFiddle](#))

```
const source = Rx.Observable.from([1,2,3,4,5]);
//no value will pass, emit default
const example = source.first(val => val > 5, val => `Value: ${val}
`, 'Nothing');
//output: 'Nothing'
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [first](#) 📖 - Official docs
- [Filtering operator: take, first, skip](#) 📺 💰 - André Staltz





Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/first.ts>

# ignoreElements

**signature:** `ignoreElements(): Observable`

**Ignore everything but complete and error.**

## Examples

**Example 1: Ignore all elements from source**

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 100ms
const source = Rx.Observable.interval(100);
//ignore everything but complete
const example = source
  .take(5)
  .ignoreElements();
//output: "COMPLETE!"
const subscribe = example.subscribe(
  val => console.log(`NEXT: ${val}`),
  val => console.log(`ERROR: ${val}`),
  () => console.log('COMPLETE!')
);
```

**Example 2: Only displaying error**

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 100ms
const source = Rx.Observable.interval(100);
//ignore everything but error
const error = source
  .flatMap(val => {
    if(val === 4){
      return Rx.Observable.throw(`ERROR AT ${val}`);
    }
    return Rx.Observable.of(val);
  })
  .ignoreElements();
//output: "ERROR: ERROR AT 4"
const subscribe = error.subscribe(
  val => console.log(`NEXT: ${val}`),
  val => console.log(`ERROR: ${val}`),
  () => console.log('SECOND COMPLETE!')
);
```

## Additional Resources

- [ignoreElements](#)  - Official docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/ignoreElements.ts>

# last

**signature:** `last(predicate: function): Observable`

**Emit the last value emitted from source on completion, based on provided expression.**

---

💡 The counterpart to last is **first**!

---

## Examples

### Example 1: Last value in sequence

([jsBin](#) | [jsFiddle](#))

```
const source = Rx.Observable.from([1, 2, 3, 4, 5]);
//no arguments, emit last value
const example = source.last();
//output: "Last value: 5"
const subscribe = example.subscribe(val => console.log(`Last value: ${val}`));
```

### Example 2: Last value to pass predicate

([jsBin](#) | [jsFiddle](#))

```
const source = Rx.Observable.from([1, 2, 3, 4, 5]);
//emit last even number
const exampleTwo = source.last(num => num % 2 === 0);
//output: "Last to pass test: 4"
const subscribeTwo = exampleTwo.subscribe(val => console.log(`Last to pass test: ${val}`));
```

### Example 3: Last with result selector

([jsBin](#) | [jsFiddle](#))

```
const source = Rx.Observable.from([1,2,3,4,5]);
//supply an option projection function for the second parameter
const exampleTwo = source.last(v => v > 4, v => `The highest emitted number was ${v}`);
//output: 'The highest emitted number was 5'
const subscribeTwo = exampleTwo.subscribe(val => console.log(val));
```

### Example 4: Last with default value

([jsBin](#) | [jsFiddle](#))

```
const source = Rx.Observable.from([1,2,3,4,5]);
//no values will pass given predicate, emit default
const exampleTwo = source.last(v => v > 5, v => v, 'Nothing!');
//output: 'Nothing!'
const subscribeTwo = exampleTwo.subscribe(val => console.log(val));
```

## Additional Resources

- [last](#) 📖 - Official docs
- [Filtering operator: takeLast, last](#) 🖥️ 💻 - André Staltz

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/last.ts>

# sample

**signature:** `sample(sampler: Observable): Observable`

**Sample from source when provided observable emits.**

## Examples

### Example 1: Sample source every 2 seconds

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 1s
const source = Rx.Observable.interval(1000);
//sample last emitted value from source every 2s
const example = source.sample(Rx.Observable.interval(2000));
//output: 2..4..6..8..
const subscribe = example.subscribe(val => console.log(val));
```

### Example 2: Sample source when interval emits

( [jsBin](#) | [jsFiddle](#) )

```
const source = Rx.Observable.zip(
  //emit 'Joe', 'Frank' and 'Bob' in sequence
  Rx.Observable.from(['Joe', 'Frank', 'Bob']),
  //emit value every 2s
  Rx.Observable.interval(2000)
);
//sample last emitted value from source every 2.5s
const example = source.sample(Rx.Observable.interval(2500));
//output: ["Joe", 0]...["Frank", 1].....
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [sample](#)  - Official docs
- 



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/sample.ts>

# single

**signature:** `single(a: Function): Observable`

**Emit single item that passes expression.**


## Examples

**Example 1: Emit first number passing predicate**

( [jsBin](#) | [jsFiddle](#) )

```
//emit (1,2,3,4,5)
const source = Rx.Observable.from([1,2,3,4,5]);
//emit one item that matches predicate
const example = source.single(val => val === 4);
//output: 4
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [single](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/single.ts>



# skip

**signature:** `skip(the: Number): Observable`

**Skip the provided number of emitted values.**

## Examples

**Example 1: Skipping values before emission**

( [jsBin](#) | [jsFiddle](#) )

```
//emit every 1s
const source = Rx.Observable.interval(1000);
//skip the first 5 emitted values
const example = source.skip(5);
//output: 5...6...7...8.....
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [skip](#) 📄 - Official docs
- [Filtering operator: take, first, skip](#) 🖥️ 💡 - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/skip.ts>

# skipUntil

**signature:** `skipUntil(the: Observable): Observable`

**Skip emitted values from source until provided observable emits.**

## Examples

**Example 1: Skip until observable emits**

( [jsBin](#) | [jsFiddle](#) )

```
//emit every 1s
const source = Rx.Observable.interval(1000);
//skip emitted values from source until inner observable emits (
6s)
const example = source.skipUntil(Rx.Observable.timer(6000));
//output: 5...6...7...8.....
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [skipUntil](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/skipUntil.ts>

# skipWhile

**signature:** `skipWhile(predicate: Function): Observable`

**Skip emitted values from source until provided expression is false.**

## Examples

**Example 1: Skip while values below threshold**

([jsBin](#) | [jsFiddle](#))

```
//emit every 1s
const source = Rx.Observable.interval(1000);
//skip emitted values from source while value is less than 5
const example = source.skipWhile(val => val < 5);
//output: 5...6...7...8.....
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [skipWhile](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/skipWhile.ts>

# take

**signature:** `take(count: number): Observable`

**Emit provided number of values before completing.**

---

💡 If you want to take a variable number of values based on some logic, or another observable, you can use [takeUntil](#) or [takeWhile](#)!

---

## Examples

### Example 1: Take 1 value from source

( [jsBin](#) | [jsFiddle](#) )




```
//emit 1,2,3,4,5
const source = Rx.Observable.of(1,2,3,4,5);
//take the first emitted value then complete
const example = source.take(1);
//output: 1
const subscribe = example.subscribe(val => console.log(val));
```

### Example 2: Take the first 5 values from source

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 1s
const interval = Rx.Observable.interval(1000);
//take the first 5 emitted values
const example = interval.take(5);
//output: 0,1,2,3,4
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [take](#)  - Official docs
  - [Filtering operator: take, first, skip](#)   - André Staltz
- 



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/take.ts>

# takeUntil

**signature:** `takeUntil(notifier: Observable): Observable`

**Emit values until provided observable emits.**

---

💡 If you only need a specific number of values, try [take](#)!

---

## Examples

### Example 1: Take values until timer emits

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 1s
const source = Rx.Observable.interval(1000);
//after 5 seconds, emit value
const timer = Rx.Observable.timer(5000);
//when timer emits after 5s, complete source
const example = source.takeUntil(timer);
//output: 0,1,2,3
const subscribe = example.subscribe(val => console.log(val));
```

### Example 2: Take the first 5 even numbers

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 1s
const source = Rx.Observable.interval(1000);
//is number even?
const isEven = val => val % 2 === 0;
//only allow values that are even
const evenSource = source.filter(isEven);
//keep a running total of the number of even numbers out
const evenNumberCount = evenSource
    .scan((acc, _) => acc + 1, 0);
//do not emit until 5 even numbers have been emitted
const fiveEvenNumbers = evenNumberCount.filter(val => val > 5);

const example = evenSource
    //also give me the current even number count for display
    .withLatestFrom(evenNumberCount)
    .map(([val, count]) => `Even number (${count}) : ${val}`)
    //when five even numbers have been emitted, complete source observable
    .takeUntil(fiveEvenNumbers);
/*
    Even number (1) : 0,
    Even number (2) : 2
    Even number (3) : 4
    Even number (4) : 6
    Even number (5) : 8
*/
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [takeUntil](#) 📄 - Official docs
- [Stopping a stream with takeUntil](#) 📄 💡 - John Linquist

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/takeUntil.ts>





# takeWhile

**signature:** `takeWhile(predicate: function(value, index): boolean): Observable`

**Emit values until provided expression is false.**

## Examples

**Example 1: Take values under limit**

( [jsBin](#) | [jsFiddle](#) )

```
//emit 1,2,3,4,5
const source = Rx.Observable.of(1,2,3,4,5);
//allow values until value from source is greater than 4, then complete
const example = source.takeWhile(val => val <= 4);
//output: 1,2,3,4
const subscribe = example.subscribe(val => console.log(val));
```

**Example 2: Difference between takeWhile() and filter()**

( [jsBin](#) | [jsFiddle](#) )

```
// emit 3, 3, 3, 9, 1, 4, 5, 8, 96, 3, 66, 3, 3, 3
const source = Rx.Observable.of(3, 3, 3, 9, 1, 4, 5, 8, 96, 3, 66,
, 3, 3, 3);



// allow values until value from source equals 3, then complete
// output: [3, 3, 3]
source
  .takeWhile(it => it === 3 )
  .subscribe(val => console.log('takeWhile', val));

// output: [3, 3, 3, 3, 3, 3, 3]
source
  .filter(it => it === 3)
  .subscribe(val => console.log('filter', 3));
```

## Related Recipes

- [Smart Counter](#)

## Additional Resources

- [takeWhile](#)  - Official docs
- [Completing a stream with takeWhile](#)   - John Linquist

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/takeWhile.ts>

# throttle

**signature:** `throttle(durationSelector: function(value): Observable | Promise): Observable`

**Emit value only when duration, determined by provided function, has passed.**

## Examples

**Example 1: Throttle for 2 seconds, based on second observable**

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 1 second
const source = Rx.Observable.interval(1000);
//throttle for 2 seconds, emit latest value
const example = source.throttle(val => Rx.Observable.interval(2000));
//output: 0...3...6...9
const subscribe = example.subscribe(val => console.log(val));
```




**Example 2: Throttle with promise**

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 1 second
const source = Rx.Observable.interval(1000);
//incrementally increase the time to resolve based on source
const promise = val => new Promise(resolve => setTimeout(() => r
resolve(`Resolved: ${val}`), val * 100));
//when promise resolves emit item from source
const example = source
    .throttle(promise)
    .map(val => `Throttled off Promise: ${val}`);

const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [throttle](#)  - Official docs
- [Filtering operator: throttle and throttleTime](#)   - André Staltz

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/throttle.ts>

# throttleTime

**signature:** `throttleTime(duration: number, scheduler: Scheduler): Observable`

**Emit latest value when specified duration has passed.**

## Examples

**Example 1: Receive latest value every 5 seconds**

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 1 second
const source = Rx.Observable.interval(1000);
/*
  throttle for five seconds
  last value emitted before throttle ends will be emitted from source
*/
const example = source
  .throttleTime(5000);
//output: 0...6...12
const subscribe = example.subscribe(val => console.log(val));
```

**Example 2: Throttle merged observable**

( [jsBin](#) | [jsFiddle](#) )

```
const source = Rx.Observable
  .merge(
    //emit every .75 seconds
    Rx.Observable.interval(750),
    //emit every 1 second
    Rx.Observable.interval(1000)
  );
//throttle in middle of emitted values
const example = source.throttleTime(1200);
//output: 0...1...4...4...8...7
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [throttleTime](#) 📄 - Official docs
- [Filtering operator: throttle and throttleTime](#) 📄 📄 - André Staltz

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/throttleTime.ts>

# Transformation Operators

Transforming values as they pass through the operator chain is a common task. These operators provide transformation techniques for nearly any use-case you will encounter.

## Contents

- [buffer](#)
- [bufferCount](#)
- [bufferTime](#) ★
- [bufferToggle](#)
- [bufferWhen](#)
- [concatMap](#) ★
- [concatMapTo](#)
- [exhaustMap](#)
- [expand](#)
- [groupBy](#)
- [map](#) ★
- [mapTo](#)
- [mergeMap](#) ★
- [partition](#)
- [pluck](#)
- [reduce](#)
- [scan](#) ★
- [switchMap](#) ★
- [window](#)
- [windowCount](#)
- [windowTime](#)
- [windowToggle](#)
- [windowWhen](#)

★ - *commonly used*





# buffer

**signature:** `buffer(closingNotifier: Observable): Observable`

**Collect output values until provided observable emits, emit as array.**




## Examples

**Example 1: Buffer until document click**

( [jsBin](#) | [jsFiddle](#) )

```
//Create an observable that emits a value every second
const myInterval = Rx.Observable.interval(1000);
//Create an observable that emits every time document is clicked
const bufferBy = Rx.Observable.fromEvent(document, 'click');
/*
Collect all values emitted by our interval observable until we c
lick document. This will cause the bufferBy Observable to emit a
value, satisfying the buffer. Pass us all collected values sinc
e last buffer as an array.
*/
const myBufferedInterval = myInterval.buffer(bufferBy);
//Print values to console
//ex. output: [1,2,3] ... [4,5,6,7,8]
const subscribe = myBufferedInterval.subscribe(val => console.log
(' Buffered Values:', val));
```

## Additional Resources

- [buffer](#)  - Official docs
- [Transformation operator: buffer](#)   - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/buffer.ts>

# bufferCount

**signature:** `bufferCount(bufferSize: number, startBufferEvery: number = null): Observable`

**Collect emitted values until provided number is fulfilled, emit as array.**

## Examples

**Example 1: Collect buffer and emit after specified number of values**

( [jsBin](#) | [jsFiddle](#) )

```
//Create an observable that emits a value every second
const source = Rx.Observable.interval(1000);
//After three values are emitted, pass on as an array of buffered values
const bufferThree = source.bufferCount(3);
//Print values to console
//ex. output [0,1,2]...[3,4,5]
const subscribe = bufferThree.subscribe(val => console.log('Buffered Values:', val));
```

**Example 2: Overlapping buffers**

( [jsBin](#) | [jsFiddle](#) )

```
//Create an observable that emits a value every second
const source = Rx.Observable.interval(1000);
/*
bufferCount also takes second argument, when to start the next b
uffer
for instance, if we have a bufferCount of 3 but second argument
(startBufferEvery) of 1:
1st interval value:
buffer 1: [0]
2nd interval value:
buffer 1: [0,1]
buffer 2: [1]
3rd interval value:
buffer 1: [0,1,2] Buffer of 3, emit buffer
buffer 2: [1,2]
buffer 3: [2]
4th interval value:
buffer 2: [1,2,3] Buffer of 3, emit buffer
buffer 3: [2, 3]
buffer 4: [3]
*/
const bufferEveryOne = source.bufferCount(3,1);
//Print values to console
const subscribe = bufferEveryOne.subscribe(val => console.log('S
tart Buffer Every 1:', val));
```

## Additional Resources

- [bufferCount](#)  - Official docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/bufferCount.ts>

# bufferTime

**signature:** `bufferTime(bufferTimeSpan: number, bufferCreationInterval: number, scheduler: Scheduler): Observable`

**Collect emitted values until provided time has passed, emit as array.**

## Examples

### Example 1: Buffer for 2 seconds

( [jsBin](#) | [jsFiddle](#) )

```
//Create an observable that emits a value every 500ms
const source = Rx.Observable.interval(500);
//After 2 seconds have passed, emit buffered values as an array
const example = source.bufferTime(2000);
//Print values to console
//ex. output [0,1,2]...[3,4,5,6]
const subscribe = example.subscribe(val => console.log('Buffered
  with Time:', val));
```

### Example 2: Multiple active buffers

( [jsBin](#) | [jsFiddle](#) )

```
//Create an observable that emits a value every 500ms
const source = Rx.Observable.interval(500);
/*
bufferTime also takes second argument, when to start the next bu
ffer (time in ms)
for instance, if we have a bufferTime of 2 seconds but second ar
gument (bufferCreationInterval) of 1 second:
ex. output: [0,1,2]...[1,2,3,4,5]...[3,4,5,6,7]
*/
const example = source.bufferTime(2000,1000);
//Print values to console
const subscribe = example.subscribe(val => console.log('Start Bu
ffer Every 1s:', val));
```

## Additional Resources

- [bufferTime](#)  - Official docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/bufferTime.ts>

# bufferToggle

**signature:** `bufferToggle(openings: Observable, closingSelector: Function): Observable`

**Toggle on to catch emitted values from source, toggle off to emit buffered values as array.**

## Examples

**Example 1: Toggle buffer on and off at interval**

([jsBin](#) | [jsFiddle](#))

```
//emit value every second
const sourceInterval = Rx.Observable.interval(1000);
//start first buffer after 5s, and every 5s after
const startInterval = Rx.Observable.interval(5000);
//emit value after 3s, closing corresponding buffer
const closingInterval = val => {
  console.log(`Value ${val} emitted, starting buffer! Closing in 3s!`)
  return Rx.Observable.interval(3000);
}
//every 5s a new buffer will start, collecting emitted values for 3s then emitting buffered values
const bufferToggleInterval = sourceInterval.bufferToggle(startInterval, closingInterval);
//log to console
//ex. emitted buffers [4,5,6]...[9,10,11]
const subscribe = bufferToggleInterval.subscribe(val => console.log('Emitted Buffer:', val));
```

## Additional Resources

- [bufferToggle](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/bufferToggle.ts>



# bufferWhen

**signature:** `bufferWhen(closingSelector: function): Observable`

**Collect all values until closing selector emits, emit buffered values.**

## Examples

**Example 1: Emit buffer based on interval**

([jsBin](#) | [jsFiddle](#))

```
//emit value every 1 second
const oneSecondInterval = Rx.Observable.interval(1000);
//return an observable that emits value every 5 seconds
const fiveSecondInterval = () => Rx.Observable.interval(5000);
//every five seconds, emit buffered values
const bufferWhenExample = oneSecondInterval.bufferWhen(fiveSecondInterval);
//log values
//ex. output: [0,1,2,3]...[4,5,6,7,8]
const subscribe = bufferWhenExample.subscribe(val => console.log('Emitted Buffer: ', val));
```

## Additional Resources

- [bufferWhen](#)  - Official docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/bufferWhen.ts>



## concatMap

**signature:** `concatMap(project: function, resultSelector: function): Observable`

**Map values to inner observable, subscribe and emit in order.**

## Examples

### Example 1: Map to inner observable

( [jsBin](#) | [jsFiddle](#) )

```
//emit 'Hello' and 'Goodbye'
const source = Rx.Observable.of('Hello', 'Goodbye');
// map value from source into inner observable, when complete emit result and move to next
const example = source.concatMap(val => Rx.Observable.of(`${val} World!`));
//output: 'Example One: 'Hello World', Example One: 'Goodbye World'
const subscribe = example
    .subscribe(val => console.log('Example One:', val));
```

### Example 2: Map to promise

( [jsBin](#) | [jsFiddle](#) )

```
//emit 'Hello' and 'Goodbye'
const source = Rx.Observable.of('Hello', 'Goodbye');
//example with promise
const examplePromise = val => new Promise(resolve => resolve(`${
val} World!`));
// map value from source into inner observable, when complete em
it result and move to next
const example = source.concatMap(val => examplePromise(val))
//output: 'Example w/ Promise: 'Hello World', Example w/ Promise
: 'Goodbye World'
const subscribe = example.subscribe(val => console.log('Example
w/ Promise:', val));
```

### Example 3: Supplying a projection function

([jsBin](#) | [jsFiddle](#))

```
//emit 'Hello' and 'Goodbye'
const source = Rx.Observable.of('Hello', 'Goodbye');
//example with promise
const examplePromise = val => new Promise(resolve => resolve(`${
val} World!`));
//result of first param passed to second param selector function
before being returned
const example = source.concatMap(val => examplePromise(val), res
ult => `${result} w/ selector!`);
//output: 'Example w/ Selector: 'Hello w/ Selector', Example w/
Selector: 'Goodbye w/ Selector'
const subscribe = example.subscribe(val => console.log('Example
w/ Selector:', val));
```




### Example 4: Illustrating difference between concatMap and mergeMap

([jsBin](#) | [jsFiddle](#))

```
const concatMapSub = Rx.Observable.of(2000, 1000)
  .concatMap(v => Rx.Observable.of(v).delay(v))
// concatMap: 2000, concatMap: 1000
  .subscribe(v => console.log('concatMap:', v))

const mergeMapSub = Rx.Observable.of(2000, 1000)
  .mergeMap(v => Rx.Observable.of(v).delay(v))
// mergeMap: 1000, mergeMap: 2000
  .subscribe(v => console.log('mergeMap:', v))
```

## Additional Resources

- [concatMap](#)  - Official docs
- [Use RxJS concatMap to map and concat higher order observables](#)   - André Staltz

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/concatMap.ts>

# concatMapTo

**signature:** `concatMapTo(observable: Observable, resultSelector: function): Observable`

**Subscribe to provided observable when previous completes, emit values.**

## Examples

### Example 1: Map to basic observable

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 2 seconds
const interval = Rx.Observable.interval(2000);
const message = Rx.Observable.of('Second(s) elapsed!');
//when interval emits, subscribe to message until complete, merge for result
const example = interval.concatMapTo(message, (time, msg) => `${time} ${msg}`);
//log values
//output: '0 Second(s) elapsed', '1 Second(s) elapsed'
const subscribe = example.subscribe(val => console.log(val));
```

### Example 2: Map to observable that emits at slower pace

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every 2 seconds
const interval = Rx.Observable.interval(2000);
//emit value every second for 5 seconds
const source = Rx.Observable.interval(1000).take(5);
/*
   ***Be Careful***: In situations like this where the source emits
   at a faster pace
   than the inner observable completes, memory issues can arise.
   (interval emits every 1 second, basicTimer completes every 5)
*/
//basicTimer will complete after 5 seconds, emitting 0,1,2,3,4
const example = interval
    .concatMapTo(source,
        (firstInterval, secondInterval) => `${firstInterval} ${secondInterval}`
    );
/*
   output: 0 0
           0 1
           0 2
           0 3
           0 4
           1 0
           1 1
           continued...
*/
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [concatMapTo](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/concatMapTo.ts>





## exhaustMap

**signature:** `exhaustMap(project: function, resultSelector: function): Observable`

**Map to inner observable, ignore other values until that observable completes.**

### Examples

**Example 1: exhaustMap with interval**

( [jsBin](#) | [jsFiddle](#) )

```
const interval = Rx.Observable.interval(1000);
const delayedInterval = interval.delay(10).take(4);

const exhaustSub = Rx.Observable
  .merge(
    // delay 10ms, then start interval emitting 4 values
    delayedInterval,
    // emit immediately
    Rx.Observable.of(true)
  )
  /*
   * The first emitted value (of(true)) will be mapped
   * to an interval observable emitting 1 value every
   * second, completing after 5.
   * Because the emissions from the delayed interval
   * fall while this observable is still active they will be ignored.
   *
   * Contrast this with concatMap which would queue,
   * switchMap which would switch to a new inner observable each emission,
   * and mergeMap which would maintain a new subscription for each emitted value.
   */
  .exhaustMap(_ => interval.take(5))
  // output: 0, 1, 2, 3, 4
  .subscribe(val => console.log(val))
```

## Example 2: Another exhaustMap with interval

( [jsBin](#) | [jsFiddle](#) )

```
const firstInterval = Rx.Observable.interval(1000).take(10);
const secondInterval = Rx.Observable.interval(1000).take(2);

const exhaustSub = firstInterval
  .do(i => console.log(`Emission of first interval: ${i}`))
  .exhaustMap(f => secondInterval)
/*
  When we subscribed to the first interval, it starts to emit a
  values (starting 0).
  This value is mapped to the second interval which then begins
  to emit (starting 0).
  While the second interval is active, values from the first int
  erval are ignored.
  We can see this when firstInterval emits number 3,6, and so on
  ...

  Output:
  Emission of first interval: 0
  0
  1
  Emission of first interval: 3
  0
  1
  Emission of first interval: 6
  0
  1
  Emission of first interval: 9
  0
  1
*/
.subscribe(s => console.log(s));
```

## Additional Resources

- [exhaustMap](#)  - Official docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/exhaustMap.ts>



## expand

**signature:** `expand(project: function, concurrent: number, scheduler: Scheduler): Observable`

**Recursively call provided function.**

## Examples

**Example 1: Add one for each invocation**

( [jsBin](#) | [jsFiddle](#) )

```
//emit 2
const source = Rx.Observable.of(2);
const example = source
  //recursively call supplied function
  .expand(val => {
    //2,3,4,5,6
    console.log(`Passed value: ${val}`);
    //3,4,5,6
    return Rx.Observable.of(1 + val);
  })
  //call 5 times
  .take(5);
/*
  "RESULT: 2"
  "Passed value: 2"
  "RESULT: 3"
  "Passed value: 3"
  "RESULT: 4"
  "Passed value: 4"
  "RESULT: 5"
  "Passed value: 5"
  "RESULT: 6"
  "Passed value: 6"
*/
//output: 2,3,4,5,6
const subscribe = example.subscribe(val => console.log(`RESULT:
${val}`));
```

## Additional Resources

- [expand](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/expand.ts>

# groupBy

**signature:** `groupBy(keySelector: Function, elementSelector: Function): Observable`

**Group into observables based on provided value.**

## Examples

### Example 1: Group by property

([jsBin](#) | [jsFiddle](#))

```
const people = [{name: 'Sue', age:25},{name: 'Joe', age: 30},{name: 'Frank', age: 25}, {name: 'Sarah', age: 35}];
//emit each person
const source = Rx.Observable.from(people);
//group by age
const example = source
  .groupBy(person => person.age)
  //return as array of each group
  .flatMap(group => group.reduce((acc, curr) => [...acc, curr], []))
  /*
  output:
  [{age: 25, name: "Sue"},{age: 25, name: "Frank"}]
  [{age: 30, name: "Joe"}]
  [{age: 35, name: "Sarah"}]
  */
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [groupBy](#)  - Official docs

- [Group higher order observables with RxJS groupBy](#) 📁 💻 - André Staltz
  - [Use groupBy in real RxJS applications](#) 📁 💻 - André Staltz
- 

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/groupBy.ts>



# map

**signature:** `map(project: Function, thisArg: any): Observable`

**Apply projection with each value from source.**

## Examples

### Example 1: Add 10 to each number

( [jsBin](#) | [jsFiddle](#) )

```
//emit (1,2,3,4,5)
const source = Rx.Observable.from([1,2,3,4,5]);
//add 10 to each value
const example = source.map(val => val + 10);
//output: 11,12,13,14,15
const subscribe = example.subscribe(val => console.log(val));
```

### Example 2: Map to single property

( [jsBin](#) | [jsFiddle](#) )

```
//emit ({name: 'Joe', age: 30}, {name: 'Frank', age: 20},{name:
'Ryan', age: 50})
const source = Rx.Observable.from([{name: 'Joe', age: 30}, {name:
'Frank', age: 20},{name: 'Ryan', age: 50}]);
//grab each persons name
const example = source.map(person => person.name);
//output: "Joe","Frank","Ryan"
const subscribe = example.subscribe(val => console.log(val));
```

## Related Recipes

- [Smart Counter](#)

## Additional Resources

- [map](#) 📄 - Official docs
- [map vs flatMap](#) 📺 - Ben Lesh
- [Transformation operator: map and mapTo](#) 📺 💰 - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/map.ts>

# mapTo

**signature:** `mapTo(value: any): Observable`

**Map emissions to constant value.**

## Examples

**Example 1: Map every emission to string**

( [jsBin](#) | [jsFiddle](#) )

```
//emit value every two seconds
const source = Rx.Observable.interval(2000);
//map all emissions to one value
const example = source.mapTo('HELLO WORLD!');
//output: 'HELLO WORLD!'...'HELLO WORLD!'...'HELLO WORLD!'...
const subscribe = example.subscribe(val => console.log(val));
```

**Example 2: Mapping clicks to string**

( [jsBin](#) | [jsFiddle](#) )

```
//emit every click on document
const source = Rx.Observable.fromEvent(document, 'click');
//map all emissions to one value
const example = source.mapTo('GOODBYE WORLD!');
//output: (click)'GOODBYE WORLD!'...
const subscribe = example.subscribe(val => console.log(val));
```

## Related Recipes

- [Smart Counter](#)

## Additional Resources

- [mapTo](#) 📄 - Official docs
  - [Changing behavior with mapTo](#) 🖨️ 💻 - John Linquist
  - [Transformation operator: map and mapTo](#) 🖨️ 💻 - André Staltz
- 

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/mapTo.ts>

# mergeMap

**signature:** `mergeMap(project: function: Observable, resultSelector: function: any, concurrent: number): Observable`

## Map to observable, emit values.

---

💡 `flatMap` is an alias for `mergeMap`!

💡 If only one inner subscription should be active at a time, try `switchMap` !

💡 If the order of emission and subscription of inner observables is important, try `concatMap` !

---

## Examples

### Example 1: mergeMap with observable

( [jsBin](#) | [jsFiddle](#) )

```
//emit 'Hello'
const source = Rx.Observable.of('Hello');
//map to inner observable and flatten
const example = source.mergeMap(val => Rx.Observable.of(`${val}
World!`));
//output: 'Hello World!'
const subscribe = example.subscribe(val => console.log(val));
```

### Example 2: mergeMap with promise

( [jsBin](#) | [jsFiddle](#) )

```
//emit 'Hello'
const source = Rx.Observable.of('Hello');
//mergeMap also emits result of promise
const myPromise = val => new Promise(resolve => resolve(`${val}
World From Promise!`));
//map to promise and emit result
const example = source.mergeMap(val => myPromise(val));
//output: 'Hello World From Promise'
const subscribe = example.subscribe(val => console.log(val));
```

### Example 3: mergeMap with resultSelector

([jsBin](#) | [jsFiddle](#))

```
/*
  you can also supply a second argument which receives the source
  value and emitted
  value of inner observable or promise
*/
//emit 'Hello'
const source = Rx.Observable.of('Hello');
//mergeMap also emits result of promise
const myPromise = val => new Promise(resolve => resolve(`${val}
World From Promise!`));
const example = source
  .mergeMap(val => myPromise(val),
    (valueFromSource, valueFromPromise) => {
      return `Source: ${valueFromSource}, Promise: ${valueFromPr
omise}`;
    });
//output: "Source: Hello, Promise: Hello World From Promise!"
const subscribe = example.subscribe(val => console.log(val));
```

### Example 4: mergeMap with concurrent value

([jsBin](#) | [jsFiddle](#))

```
//emit value every 1s
const source = Rx.Observable.interval(1000);

const example = source.mergeMap(
  //project
  val => Rx.Observable.interval(5000).take(2),
  //resultSelector
  (oVal, iVal, oIndex, iIndex) => [oIndex, oVal, iIndex, iVal],
  //concurrent
  2
);
/*
    Output:
    [0, 0, 0, 0] <--1st inner observable
    [1, 1, 0, 0] <--2nd inner observable
    [0, 0, 1, 1] <--1st inner observable
    [1, 1, 1, 1] <--2nd inner observable
    [2, 2, 0, 0] <--3rd inner observable
    [3, 3, 0, 0] <--4th inner observable
*/
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [mergeMap](#) 📄 - Official docs
- [map vs flatMap](#) 📺 📄 - Ben Lesh
- [Async requests and responses in RxJS](#) 📺 📄 - André Staltz
- [Use RxJS mergeMap to map and merge higher order observables](#) 📺 📄 - André Staltz
- [Use RxJS mergeMap for fine grain custom behavior](#) 📺 📄 - André Staltz

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/mergeMap.ts>

## partition

**signature:** `partition(predicate: function: boolean, thisArg: any): [Observable, Observable]`

**Split one observable into two based on provided predicate.**

## Examples

### Example 1: Split even and odd numbers

( [jsBin](#) | [jsFiddle](#) )

```
const source = Rx.Observable.from([1,2,3,4,5,6]);
//first value is true, second false
const [evens, odds] = source.partition(val => val % 2 === 0);
/*
  Output:
  "Even: 2"
  "Even: 4"
  "Even: 6"
  "Odd: 1"
  "Odd: 3"
  "Odd: 5"
*/
const subscribe = Rx.Observable.merge(
  evens
    .map(val => `Even: ${val}`),
  odds
    .map(val => `Odd: ${val}`)
).subscribe(val => console.log(val));
```

### Example 2: Split success and errors

( [jsBin](#) | [jsFiddle](#) )



```
const source = Rx.Observable.from([1, 2, 3, 4, 5, 6]);
//if greater than 3 throw
const example = source
  .map(val => {
    if(val > 3){
      throw `${val} greater than 3!`
    }
    return {success: val};
  })
  .catch(val => Rx.Observable.of({error: val}));
//split on success or error
const [success, error] = example.partition(res => res.success)
/*
  Output:
  "Success! 1"
  "Success! 2"
  "Success! 3"
  "Error! 4 greater than 3!"
*/
const subscribe = Rx.Observable.merge(
  success.map(val => `Success! ${val.success}`),
  error.map(val => `Error! ${val.error}`)
).subscribe(val => console.log(val));
```

## Additional Resources

- [partition](#)  - Official docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/partition.ts>

# pluck

**signature:** `pluck(properties: ...args): Observable`

**Select properties to emit.**

## Examples

### Example 1: Pluck object property

( [jsBin](#) | [jsFiddle](#) )


```
const source = Rx.Observable.from([
  {name: 'Joe', age: 30},
  {name: 'Sarah', age: 35}
]);
//grab names
const example = source.pluck('name');
//output: "Joe", "Sarah"
const subscribe = example.subscribe(val => console.log(val));
```

### Example 2: Pluck nested properties

( [jsBin](#) | [jsFiddle](#) )

```
const source = Rx.Observable.from([
  {name: 'Joe', age: 30, job: {title: 'Developer', language: 'JavaScript'}},
  //will return undefined when no job is found
  {name: 'Sarah', age: 35}
]);
//grab title property under job
const example = source.pluck('job', 'title');
//output: "Developer" , undefined
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [pluck](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/pluck.ts>

# reduce

**signature:** `reduce(accumulator: function, seed: any): Observable`

**Reduces the values from source observable to a single value that's emitted when the source completes.**

💡 Just like `Array.prototype.reduce()`

💡 If you need the current accumulated value on each emission, try [scan](#)!

## Examples

### Example 1: Sum a stream of numbers

([jsBin](#) | [jsFiddle](#))

```
const source = Rx.Observable.of(1, 2, 3, 4);
const example = source.reduce((acc, val) => acc + val);
//output: Sum: 10'
const subscribe = example.subscribe(val => console.log('Sum:', val));
```

## Additional Resources

- [reduce](#) 📄 - Official docs
- [Scan\(\) vs reduce\(\) | RxJS TUTORIAL](#) 📺 - Academind

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/reduce.ts>



## scan

**signature:** `scan(accumulator: function, seed: any): Observable`

## Reduce over time.

---

💡 This operator is the core for many RxJS based [Redux](#) implementations!

---

## Examples

### Example 1: Sum over time

( [jsBin](#) | [jsFiddle](#) )

```
const subject = new Rx.Subject();
//basic scan example, sum over time starting with zero
const example = subject
  .startWith(0)
  .scan((acc, curr) => acc + curr);
//log accumulated values
const subscribe = example.subscribe(val => console.log('Accumulated total:', val));
//next values into subject, adding to the current sum
subject.next(1); //1
subject.next(2); //3
subject.next(3); //6
```

### Example 2: Accumulating an object

( [jsBin](#) | [jsFiddle](#) )

```
const subject = new Rx.Subject();
//scan example building an object over time
const example = subject.scan((acc, curr) => Object.assign({}, acc, curr), {});
//log accumulated values
const subscribe = example.subscribe(val => console.log('Accumulated object:', val));
//next values into subject, adding properties to object
subject.next({name: 'Joe'}); // {name: 'Joe'}
subject.next({age: 30}); // {name: 'Joe', age: 30}
subject.next({favoriteLanguage: 'JavaScript'}); // {name: 'Joe', age: 30, favoriteLanguage: 'JavaScript'}
```

### Example 3: Emitting random values from the accumulated array.

( [jsBin](#) | [jsFiddle](#) )

```
// Accumulate values in an array, emit random values from this array.
const scanObs = Rx.Observable.interval(1000)
  .scan((a, c) => a.concat(c), [])
  .map(r => r[Math.floor(Math.random()*r.length)])
  .distinctUntilChanged()
  .subscribe(console.log);
```

## Related Recipes

- [Smart Counter](#)

## Additional Resources

- [scan](#) 📖 - Official docs
- [Aggregating streams with reduce and scan using RxJS](#) 📖 - Ben Lesh
- [Updating data with scan](#) 📖 📄 - John Linquist
- [Transformation operator: scan](#) 📖 📄 - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/scan.ts>



# switchMap

**signature:** `switchMap(project: function: Observable, resultSelector: function(outerValue, innerValue, outerIndex, innerIndex): any): Observable`

**Map to observable, complete previous inner observable, emit values.**

---

💡 If you would like more than one inner subscription to be maintained, try `mergeMap` !

💡 This operator is generally considered a safer default to `mergeMap` !

💡 This operator can cancel in-flight network requests!

---

## Examples

### Example 1: Restart interval every 5 seconds

( [jsBin](#) | [jsFiddle](#) )

```
//emit immediately, then every 5s
const source = Rx.Observable.timer(0, 5000);
//switch to new inner observable when source emits, emit items that are emitted
const example = source.switchMap(() => Rx.Observable.interval(500));
//output: 0,1,2,3,4,5,6,7,8,9...0,1,2,3,4,5,6,7,8
const subscribe = example.subscribe(val => console.log(val));
```

### Example 2: Reset on every click

( [jsBin](#) | [jsFiddle](#) )

```
//emit every click
const source = Rx.Observable.fromEvent(document, 'click');
//if another click comes within 3s, message will not be emitted
const example = source.switchMap(val => Rx.Observable.interval(3000).mapTo('Hello, I made it!'));
// (click)...3s...'Hello I made it!'...(click)...2s(click)...
const subscribe = example.subscribe(val => console.log(val));
```

### Example 3: Using a `resultSelector` function

( [jsBin](#) | [jsFiddle](#) )

```
//emit immediately, then every 5s
const source = Rx.Observable.timer(0, 5000);
//switch to new inner observable when source emits, invoke project function and emit values
const example = source.switchMap(() => Rx.Observable.interval(2000), (outerValue, innerValue, outerIndex, innerIndex) => ({outerValue, innerValue, outerIndex, innerIndex}));
/*
    Output:
    {outerValue: 0, innerValue: 0, outerIndex: 0, innerIndex: 0}
    {outerValue: 0, innerValue: 1, outerIndex: 0, innerIndex: 1}
    {outerValue: 1, innerValue: 0, outerIndex: 1, innerIndex: 0}
    {outerValue: 1, innerValue: 1, outerIndex: 1, innerIndex: 1}
*/
const subscribe = example.subscribe(val => console.log(val));
```

### Example 4: Countdown timer with `switchMap`

( [jsBin](#) | [jsFiddle](#) )

```
const countdownSeconds = 60;
const setHTML = id => val => document.getElementById(id).innerHTML = val;
const pauseButton = document.getElementById('pause');
const resumeButton = document.getElementById('resume');
const interval$ = Rx.Observable.interval(1000).mapTo(-1);

const pause$ = Rx.Observable.fromEvent(pauseButton, 'click').mapTo(Rx.Observable.of(false))
const resume$ = Rx.Observable.fromEvent(resumeButton, 'click').mapTo(interval$);

const timer$ = Rx.Observable
  .merge(pause$, resume$)
  .startWith(interval$)
  .switchMap(val => val)
  // if pause button is clicked stop countdown
  .scan((acc, curr) => curr ? curr + acc : acc, countdownSeconds)

  .subscribe(setHTML('remaining'));
```








## HTML

```
<h4>
Time remaining: <span id="remaining"></span>
</h4>
<button id="pause">
Pause Timer
</button>
<button id="resume">
Resume Timer
</button>
```

## Related Recipes

- [Smart Counter](#)

## Additional Resources

- [switchMap](#)  - Official docs
  - [Starting a stream with switchMap](#)   - John Linquist
  - [Use RxJS switchMap to map and flatten higher order observables](#)   - André Staltz
  - [Use switchMap as a safe default to flatten observables in RxJS](#)   - André Staltz
- 

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/switchMap.ts>

# window

**signature:** `window(windowBoundaries: Observable): Observable`

**Observable of values for window of time.**

## Examples

**Example 1: Open window specified by inner observable**

( [jsBin](#) | [jsFiddle](#) )

```
//emit immediately then every 1s
const source = Rx.Observable.timer(0, 1000);
const example = source
    .window(Rx.Observable.interval(3000))
const count = example.scan((acc, curr) => acc + 1, 0)
/*
    "Window 1:"
    0
    1
    2
    "Window 2:"
    3
    4
    5
    ...
*/
const subscribe = count.subscribe(val => console.log(`Window ${v
al}:`));
const subscribeTwo = example.mergeAll().subscribe(val => console.
log(val));
```

## Additional Resources

- [window](#)  - Official docs

- [Split an RxJS observable with window](#)   - André Staltz
- 

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/window.ts>

# windowCount

**signature:** `windowCount(windowSize: number, startWindowEvery: number): Observable`

**Observable of values from source, emitted each time provided count is fulfilled.**

## Examples

**Example 1: Start new window every x items emitted**

( [jsBin](#) | [jsFiddle](#) )

```
//emit every 1s
const source = Rx.Observable.interval(1000);
const example = source
    //start new window every 4 emitted values
    .windowCount(4)
    .do(() => console.log('NEW WINDOW!'))

const subscribeTwo = example
    //window emits nested observable
    .mergeAll()
/*
output:
"NEW WINDOW!"
0
1
2
3
"NEW WINDOW!"
4
5
6
7
*/
.subscribe(val => console.log(val));
```

## Additional Resources

- [windowCount](#)  - Official docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/windowCount.ts>



# windowTime

**signature:** `windowTime(windowTimeSpan: number, windowCreationInterval: number, scheduler: Scheduler): Observable`

**Observable of values collected from source for each provided time span.**

## Examples

**Example 1: Open new window every specified duration**

( [jsBin](#) | [jsFiddle](#) )

```
//emit immediately then every 1s
const source = Rx.Observable.timer(0,1000);
const example = source
    //start new window every 3s
    .windowTime(3000)
    .do(() => console.log('NEW WINDOW!'))

const subscribeTwo = example
    //window emits nested observable
    .mergeAll()
/*
output:
"NEW WINDOW!"
0
1
2
"NEW WINDOW!"
3
4
5
*/
.subscribe(val => console.log(val));
```

## Additional Resources

- [windowTime](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/windowTime.ts>

# windowToggle

**signature:** `windowToggle(openings: Observable,  
closingSelector: function(value): Observable): Observable`

**Collect and emit observable of values from source between opening and closing emission.**

## Examples

**Example 1: Toggle window at increasing interval**

( [jsBin](#) | [jsFiddle](#) )

```
//emit immediately then every 1s
const source = Rx.Observable.timer(0,1000);
//toggle window on every 5
const toggle = Rx.Observable.interval(5000);
const example = source
    //turn window on every 5s
    .windowToggle(toggle, (val) => Rx.Observable.interval(val *
1000))
    .do(() => console.log('NEW WINDOW!'))

const subscribeTwo = example
    //window emits nested observable
    .mergeAll()
/*
output:
"NEW WINDOW!"
5
"NEW WINDOW!"
10
11
"NEW WINDOW!"
15
16
"NEW WINDOW!"
20
21
22
*/
.subscribe(val => console.log(val));
```

## Additional Resources

- [windowToggle](#) 📄 - Official docs
- [Split an RxJS observable conditionally with windowToggle](#) 📄 💻 - André Staltz

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/windowToggle.ts>



# windowWhen

**signature:** `windowWhen(closingSelector: function(): Observable): Observable`

**Close window at provided time frame emitting observable of collected values from source.**

## Examples

**Example 1: Open and close window at interval**

( [jsBin](#) | [jsFiddle](#) )

```
//emit immediately then every 1s
const source = Rx.Observable.timer(0,1000);
const example = source
    //close window every 5s and emit observable of collected values from source
    .windowWhen((val) => Rx.Observable.interval(5000))
    .do(() => console.log('NEW WINDOW!'))

const subscribeTwo = example
    //window emits nested observable
    .mergeAll()
/*
output:
"NEW WINDOW!"
0
1
2
3
4
"NEW WINDOW!"
5
6
7
8
9
*/
.subscribe(val => console.log(val));
```

## Additional Resources

- [windowWhen](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/windowWhen.ts>

# Utility Operators

From logging, handling notifications, to setting up schedulers, these operators provide helpful utilities in your observable toolkit.

## Contents

- [do](#) ★
- [delay](#)
- [delayWhen](#)
- [dematerialize](#)
- [let](#)
- [toPromise](#)

★ - *commonly used*



# do

**signature:** `do(nextOrObserver: function, error: function, complete: function): Observable`

**Transparently perform actions or side-effects, such as logging.**

## Examples

### Example 1: Logging with do

([jsBin](#) | [jsFiddle](#))

```
const source = Rx.Observable.of(1,2,3,4,5);
//transparently log values from source with 'do'
const example = source
  .do(val => console.log(`BEFORE MAP: ${val}`))
  .map(val => val + 10)
  .do(val => console.log(`AFTER MAP: ${val}`));
//'do' does not transform values
//output: 11...12...13...14...15
const subscribe = example.subscribe(val => console.log(val));
```

## Additional Resources

- [do](#) 📄 - Official docs
- [Logging a stream with do](#) 🖥️ 💻 - John Linquist
- [Utility operator: do](#) 🖥️ 💻 - André Staltz

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/do.ts>



# delay

**signature:** `delay(delay: number | Date, scheduler: Scheduler): Observable`

**Delay emitted values by given time.**

## Examples

**Example 1: Delay for increasing durations**

([jsBin](#) | [jsFiddle](#))

```
//emit one item
const example = Rx.Observable.of(null);
//delay output of each by an extra second
const message = Rx.Observable.merge(
  example.mapTo('Hello'),
  example.mapTo('World!').delay(1000),
  example.mapTo('Goodbye').delay(2000),
  example.mapTo('World!').delay(3000)
);
//output: 'Hello'...'World!'...'Goodbye'...'World!'
const subscribe = message.subscribe(val => console.log(val));
```

## Additional Resources

- [delay](#) 📖 - Official docs
- [Transformation operator: delay and delayWhen](#) 🖥️ 💡 - André Staltz

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/delay.ts>



# delayWhen

**signature:** `delayWhen(selector: Function, sequence: Observable): Observable`

**Delay emitted values determined by provided function.**

## Examples

**Example 1: Delay based on observable**

([jsBin](#) | [jsFiddle](#))

```
//emit value every second
const message = Rx.Observable.interval(1000);
//emit value after five seconds
const delayForFiveSeconds = () => Rx.Observable.timer(5000);
//after 5 seconds, start emitting delayed interval values
const delayWhenExample = message.delayWhen(delayForFiveSeconds);
//log values, delayed for 5 seconds
//ex. output: 5s....1...2...3
const subscribe = delayWhenExample.subscribe(val => console.log(val));
```

## Additional Resources

- [delayWhen](#) 📖 - Official docs
- [Transformation operator: delay and delayWhen](#) 🖨️ 💡 - André Staltz

---

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/delayWhen.ts>



# dematerialize

**signature:** `dematerialize(): Observable`

**Turn notification objects into notification values.**

## Examples

**Example 1: Converting notifications to values**

([jsBin](#) | [jsFiddle](#))

```
//emit next and error notifications
const source = Rx.Observable
  .from([
    Rx.Notification.createNext('SUCCESS!'),
    Rx.Notification.createError('ERROR!')
  ])
//turn notification objects into notification values
  .dematerialize();

//output: 'NEXT VALUE: SUCCESS' 'ERROR VALUE: 'ERROR!'
const subscription = source.subscribe({
  next: val => console.log(`NEXT VALUE: ${val}`),
  error: val => console.log(`ERROR VALUE: ${val}`)
});
```

## Additional Resources

- [dematerialize](#)  - Official docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/dematerialize.ts>





# let

**signature:** `let(function): Observable`

**Let me have the whole observable.**

## Examples

**Example 1: Reusing error handling logic with let**

( [jsBin](#) | [jsFiddle](#) )

```
// custom error handling logic
const retryThreeTimes = obs => obs.retry(3).catch(_ => Rx.Observable.of('ERROR!'));
const examplePromise = val => new Promise(resolve => resolve(`Complete: ${val}`));

//faking request
const subscribe = Rx.Observable
  .of('some_url')
  .mergeMap(url => examplePromise(url))
  // could reuse error handling logic in multiple places with let

  .let(retryThreeTimes)
  //output: Complete: some_url
  .subscribe(result => console.log(result));

const customizableRetry = retryTimes => obs => obs.retry(retryTimes).catch(_ => Rx.Observable.of('ERROR!'));

//faking request
const secondSubscribe = Rx.Observable
  .of('some_url')
  .mergeMap(url => examplePromise(url))
  // could reuse error handling logic in multiple places with let

  .let(customizableRetry(3))
  //output: Complete: some_url
  .subscribe(result => console.log(result));
```

## Example 2: Applying map with let

( [jsBin](#) | [jsFiddle](#) )

```
//emit array as a sequence
const source = Rx.Observable.from([1,2,3,4,5]);
//demonstrating the difference between let and other operators
const test = source
  .map(val => val + 1)
  /*
    this would fail, let behaves differently than most operators
    val in this case is an observable
  */
  //.let(val => val + 2)
  .subscribe(val => console.log('VALUE FROM ARRAY: ', val));

const subscribe = source
  .map(val => val + 1)
  //'let' me have the entire observable
  .let(obs => obs.map(val => val + 2))
  //output: 4,5,6,7,8
  .subscribe(val => console.log('VALUE FROM ARRAY WITH let: ', val));
```

### Example 3: Applying multiple operators with let

([jsBin](#) | [jsFiddle](#))

```
//emit array as a sequence
const source = Rx.Observable.from([1,2,3,4,5]);

//let provides flexibility to add multiple operators to source observable then return
const subscribeTwo = source
  .map(val => val + 1)
  .let(obs => obs
    .map(val => val + 2)
    //also, just return evens
    .filter(val => val % 2 === 0)
  )
  //output: 4,6,8
  .subscribe(val => console.log('let WITH MULTIPLE OPERATORS: ', val));
```

## Example 4: Applying operators through function

([jsBin](#) | [jsFiddle](#))

```
//emit array as a sequence
const source = Rx.Observable.from([1,2,3,4,5]);

//pass in your own function to add operators to observable
const obsArrayPlusYourOperators = (yourAppliedOperators) => {
  return source
    .map(val => val + 1)
    .let(yourAppliedOperators)
};
const addTenThenTwenty = obs => obs.map(val => val + 10).map(val
=> val + 20);
const subscribe = obsArrayPlusYourOperators(addTenThenTwenty)
  //output: 32, 33, 34, 35, 36
  .subscribe(val => console.log('let FROM FUNCTION:', val));
```

## Additional Resources

- [let](#)  - Official docs

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/let.ts>

# toPromise

**signature:** `toPromise() : Promise`

## Convert observable to promise.

### Examples

#### Example 1: Basic Promise

( [jsBin](#) | [jsFiddle](#) )

```
//return basic observable
const sample = val => Rx.Observable.of(val).delay(5000);
//convert basic observable to promise
const example = sample('First Example')
  .toPromise()
  //output: 'First Example'
  .then(result => {
    console.log('From Promise:', result);
  });
```

#### Example 2: Using Promise.all

( [jsBin](#) | [jsFiddle](#) )

```
//return basic observable
const sample = val => Rx.Observable.of(val).delay(5000);
/*
  convert each to promise and use Promise.all
  to wait for all to resolve
*/
const example = () => {
  return Promise.all([
    sample('Promise 1').toPromise(),
    sample('Promise 2').toPromise()
  ]);
}
//output: ["Promise 1", "Promise 2"]
example().then(val => {
  console.log('Promise.all Result:', val);
});
```

## Additional Resources

- [toPromise](#)  - Official Docs

---

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/operator/toPromise.ts>

# RxJS 5 Operators By Example

A complete list of RxJS 5 operators with clear explanations, relevant resources, and executable examples.

*Prefer a split by operator type?*

## Contents (In Alphabetical Order)

- [buffer](#)
- [bufferCount](#)
- [bufferTime](#) ★
- [bufferToggle](#)
- [bufferWhen](#)
- [catch](#) ★
- [combineAll](#)
- [combineLatest](#) ★
- [concat](#) ★
- [concatAll](#)
- [concatMap](#) ★
- [concatMapTo](#)
- [create](#)
- [debounce](#)
- [debounceTime](#) ★
- [defaultIfEmpty](#)
- [distinctUntilChanged](#) ★
- [delay](#)
- [delayWhen](#)
- [do](#) ★
- [every](#)
- [empty](#)
- [expand](#)
- [exhaustMap](#)
- [filter](#) ★
- [first](#)

- `forkJoin`
- `from` ★
- `fromEvent`
- `fromPromise` ★
- `groupBy`
- `ignoreElements`
- `interval`
- `last`
- `let`
- `map` ★
- `mapTo`
- `merge` ★
- `mergeAll`
- `mergeMap` ★
- `multicast`
- `of` ★
- `partition`
- `pluck`
- `publish`
- `race`
- `range`
- `retry`
- `retryWhen`
- `sample`
- `share` ★
- `single`
- `skip`
- `skipUntil`
- `skipWhile`
- `startWith` ★
- `take` ★
- `takeUntil` ★
- `takeWhile`
- `throttle`
- `throttleTime`
- `throw`



- [timer](#)
- [toPromise](#)
- [scan](#) ★
- [switchMap](#) ★
- [window](#)
- [windowCount](#)
- [windowTime](#)
- [windowToggle](#)
- [windowWhen](#)
- [withLatestFrom](#) ★
- [zip](#)

★ - *commonly used*

## Additional Resources

- [What Are Operators?](#) 📄 - Official Docs
- [What Operators Are](#) 🖥️💡 - André Staltz

# Recipes

Common use-cases and interesting recipes to help learn RxJS.

## Contents

- [Smart Counter](#)

## Smart Counter

An interesting element on interfaces which involve dynamically updating numbers is a smart counter, or odometer effect. Instead of jumping a number up and down, quickly counting to the desired number can achieve a cool effect. An example of a popular library that accomplishes this is [odometer](#) by [Hubspot](#). Let's see how we can accomplish something similar with just a few lines of RxJS.

### Vanilla JS

( [JSBin](#) | [JSFiddle](#) )

```
// utility functions
const takeUntilFunc = (endRange, currentNumber) => {
  return endRange > currentNumber
    ? val => val <= endRange
    : val => val >= endRange;
};

const positiveOrNegative = (endRange, currentNumber) => {
  return endRange > currentNumber ? 1 : -1;
};

const updateHTML = id => val => document.getElementById(id).innerHTML = val;
// display
const input = document.getElementById('range');
const updateButton = document.getElementById('update');

const subscription = (function(currentNumber) {
  return Rx.Observable
    .fromEvent(updateButton, 'click')
    .map(_ => parseInt(input.value))
    .switchMap(endRange => {
      return Rx.Observable.timer(0, 20)
        .mapTo(positiveOrNegative(endRange, currentNumber))
        .startWith(currentNumber)
        .scan((acc, curr) => acc + curr)
        // .delayWhen(//easing here)
        .takeWhile(takeUntilFunc(endRange, currentNumber))
    })
    .do(v => currentNumber = v)
    .startWith(currentNumber)
    .subscribe(updateHTML('display'));
})(0));
```

## HTML

```
<input id="range" type="number">
<button id="update">Update</button>
<h3 id="display">0</h3>
```

We can easily take our vanilla smart counter and wrap it in any popular component based UI library. Below is an example of an Angular smart counter component which takes an `Input` of the updated end ranges and performs the appropriate transition.

## Angular Version

```
@Component({
  selector: 'number-tracker',
  template: `
    <h3> {{ currentNumber }}</h3>
  `
})
export class NumberTrackerComponent implements OnDestroy {
  @Input()
  set end(endRange: number) {
    this._counterSub$.next(endRange);
  }
  public currentNumber = 0;
  private _counterSub$ = new Subject();
  private _subscription : Subscription;

  constructor() {
    this._subscription = this._counterSub$
      .switchMap(endRange => {
        return timer(0, 20)
          .mapTo(this.positiveOrNegative(endRange, this.currentNumber))
          .startWith(this.currentNumber)
          .scan((acc, curr) => acc + curr)
          // .delayWhen(i => {
          //   easing here
          // })
          .takeWhile(this.takeUntilFunc(endRange, this.currentNumber));
      })
      .subscribe(val => this.currentNumber = val);
  }

  private positiveOrNegative(endRange, currentNumber) {
```

```
        return endRange > currentNumber ? 1 : -1;
    }

    private takeUntilFunc(endRange, currentNumber) {
        return endRange > currentNumber
            ? val => val <= endRange
            : val => val >= endRange;
    }

    ngOnDestroy() {
        this._subscription.unsubscribe();
    }
}
```

## Operators Used

- fromEvent
- map
- mapTo
- scan
- startWith
- switchMap
- takeWhile

# Concepts

Short explanations of common RxJS scenarios and use-cases.

## Contents

- [Understanding Operator Imports](#)

# Understanding Operator Imports

A problem you may have run into in the past when consuming or creating a public library that depends on RxJS is handling operator inclusion. The most predominant way to include operators in your project is to import them like below:

```
import 'rxjs/add/operator/take';
```

This adds the imported operator to the `Observable` prototype for use throughout your project:

[\(Source\)](#)

```
import { Observable } from '../Observable';
import { take } from '../operator/take';

Observable.prototype.take = take;

declare module '../Observable' {
  interface Observable<T> {
    take: typeof take;
  }
}
```

This method is generally *OK* for private projects and modules, the issue arises when you are using these imports in say, an [npm](#) package or library to be consumed throughout your organization.

## A Quick Example

To see where a problem can spring up, let's imagine **Person A** is creating a public Angular component library. In this library you need a few operators so you add the typical imports:

*some-public-library.ts*



```
import 'rxjs/add/operator/take';  
import 'rxjs/add/operator/concatMap';  
import 'rxjs/add/operator/switchMap';
```

**Person B** comes along and includes your library. They now have access to these operators even though they did not personally import them. *Probably not a huge deal but it can be confusing.* You use the library and operators, life goes on...

A month later **Person A** decides to update their library. They no longer need

`switchMap` or `concatMap` so they remove the imports:

*some-public-library.ts*

```
import 'rxjs/add/operator/take';
```

**Person B** upgrades the dependency, builds their project, which now fails. They never included `switchMap` or `concatMap` themselves, it was **just working** based on the inclusion of a 3rd party dependency. If you were not aware this could be an issue it may take a bit to track down.

## The Solution

Instead of importing operators like:

```
import 'rxjs/add/operator/take';
```

We can instead import them like:

```
import { take } from 'rxjs/operator/take';
```

This keeps them off the `Observable` prototype and let's us call them directly:

```
import { take } from 'rxjs/operator/take';
import { of } from 'rxjs/observable/of';

take.call(
  of(1, 2, 3),
  2
);
```

This quickly gets **ugly** however, imagine we have a longer chain:

```
import { take } from 'rxjs/operator/take';
import { map } from 'rxjs/operator/map';
import { of } from 'rxjs/observable/of';

map.call(
  take.call(
    of(1, 2, 3),
    2
  ),
  val => val + 2
);
```

Pretty soon we have a block of code that is near impossible to understand. How can we get the best of both worlds?

## RxJS Helpers

RxJS now comes with a `pipe` helper on `Observable` that alleviates the pain of not having operators on the prototype. We can take the ugly block of code from above:

```
import { take } from 'rxjs/operator/take';
import { map } from 'rxjs/operator/map';
import { of } from 'rxjs/observable/of';

map.call(
  take.call(
    of(1, 2, 3),
    2
  ),
  val => val + 2
);
```

And transform it into:

```
import { take, map } from 'rxjs/operators';
import { of } from 'rxjs/observable/of';

of(1, 2, 3)
  .pipe(
    take(2),
    map(val => val + 2)
  );
```

Much easier to read, right? If you are developing in Angular and using the `@angular/cdk` you can utilize the `RxChain` function and helper methods right now for a similar feel:

```
import { of } from 'rxjs/observable/of';
import { RxChain, map, debounceTime } from '@angular/cdk/rxjs';

RxChain
  .from(of(1, 2, 3))
  .call(map, val => val + 2)
  .call(debounceTime, 1000)
```

Whichever approach you prefer, both make it easy for you to keep a readable operator chain while maintaining a clean prototype for other projects!

