
OPERATORS IN RXJS
OPERATORS YOU ACTUALLY CARE ABOUT EXPLAINED FOR REAL WORLD USE CASES

TRACY LEE

- Lead, This Dot Labs, JS consulting (Angular, React, Vue, Node, Polymer)
- RxJS Core Team
- Google Developer Expert, Angular
- Microsoft MVP
- Community Rel, Node.js @ OpenJS Foundation
- Women Techmakers, GDG Silicon Valley & Triangle
- Modern Web Podcast

@ladyleet

THE ANATOMY OF AN OPERATOR

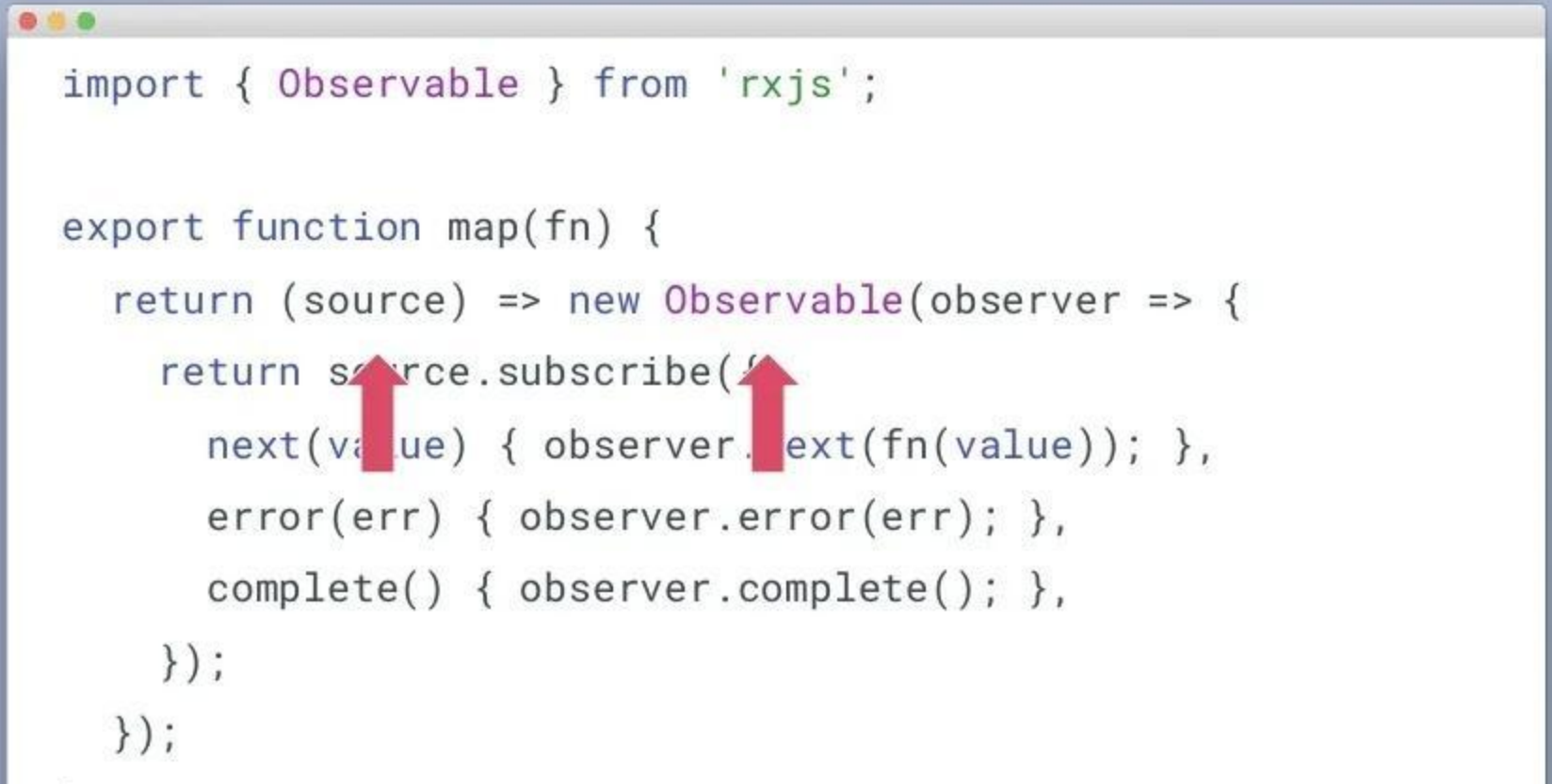
“operators” in Observables

```
import { of as observableOf } from 'rxjs';
import { map, filter } from 'rxjs/operators';

const src = observableOf(1, 2, 3, 4, 5, 6, 7);

src.pipe(
  filter(x => x % 2 === 0),
  map(x => x + x),
).subscribe(x => console.log(x)); // 4...8...12...
```

A simple map operator implementation

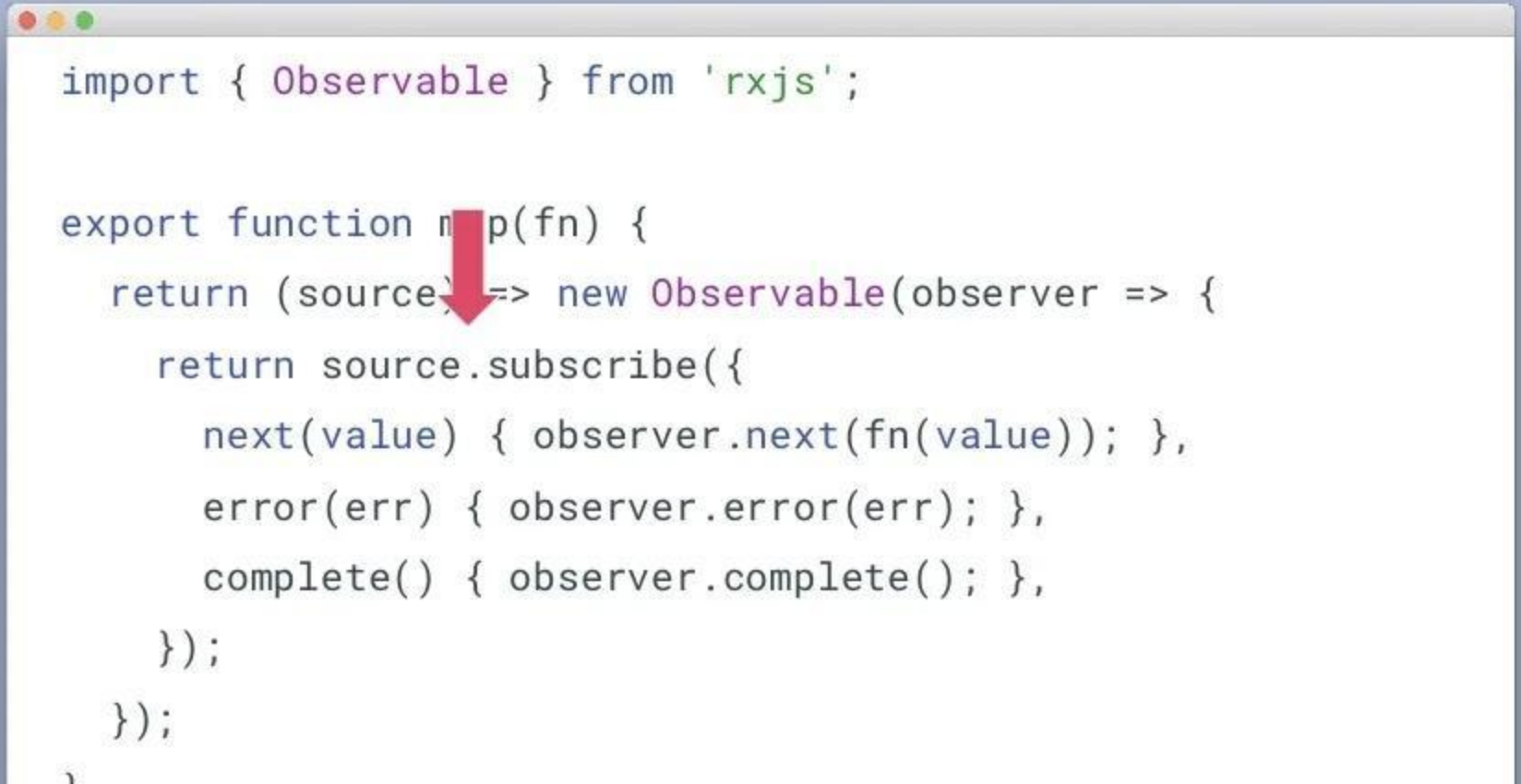
```
import { Observable } from 'rxjs';

export function map(fn) {
  return (source) => new Observable(observer => {
    return source.subscribe({
      next(value) { observer.next(fn(value)); },
      error(err) { observer.error(err); },
      complete() { observer.complete(); },
    });
  });
```

Takes an observable & returns a new one



```
import { Observable } from 'rxjs';

export function map(fn) {
  return (source) => new Observable(observer => {
    return source.subscribe({
      next(value) { observer.next(fn(value)); },
      error(err) { observer.error(err); },
      complete() { observer.complete(); },
    });
  });
```

Subscribes to the source



Passes along the transformed value

```
import { Observable } from 'rxjs';

export function map(fn) {
  return (source) => new Observable(observer => {
    return source.subscribe({
      next(value) { observer.next(fn(value)); },
      error(err) { observer.error(err); },
      complete() { observer.complete(); },
    });
  });
```

And sending along the other signals

```
import { Observable } from 'rxjs';

export function map(fn) {
  return (source) => new Observable(observer => {
    return source.subscribe({
      next(value) { observer.next(fn(value)); },
      error(err) { observer.error(err); },
      complete() { observer.complete(); },
    });
  });
}
```

There are 60+ operators in RxJS

- Flattening Operators
- Error Handling
- Completion Operators
- Subjects & Multicasting


THE EASIEST
OPERATORS
map
filter
scan


THE EASIEST
OPERATORS
map
filter
scan

map

EXPLANATION -

Most common operator in apps.

Simply meant for transforming data.

PRIMARY USES -

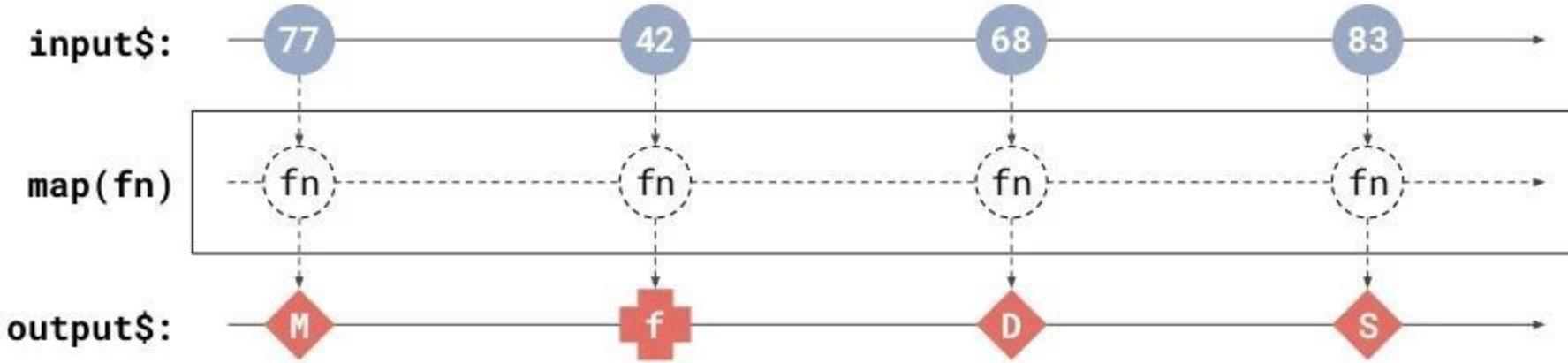
Used when a stream of data isn't the right shape and you want t
change it to another shape.

https://rxjs.dev/api/operators

map

: string : boolean : number

```
fn:  v => (v < 65 || v > 91) ? false : String.fromCharCode(v)
```



map example

```
// RxJS v6+
// Map every click to the clientX position of that click

import { fromEvent } from 'rxjs';
import { map } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const positions = clicks.pipe(map(ev => ev.clientX));


positions.subscribe(x => console.log(x));
```

map

COMMON MISTAKES - Over engineering!

When you have an array and wrap it in an observable just to use simple operators like map, reduce, or filter, you might be overthinking it.

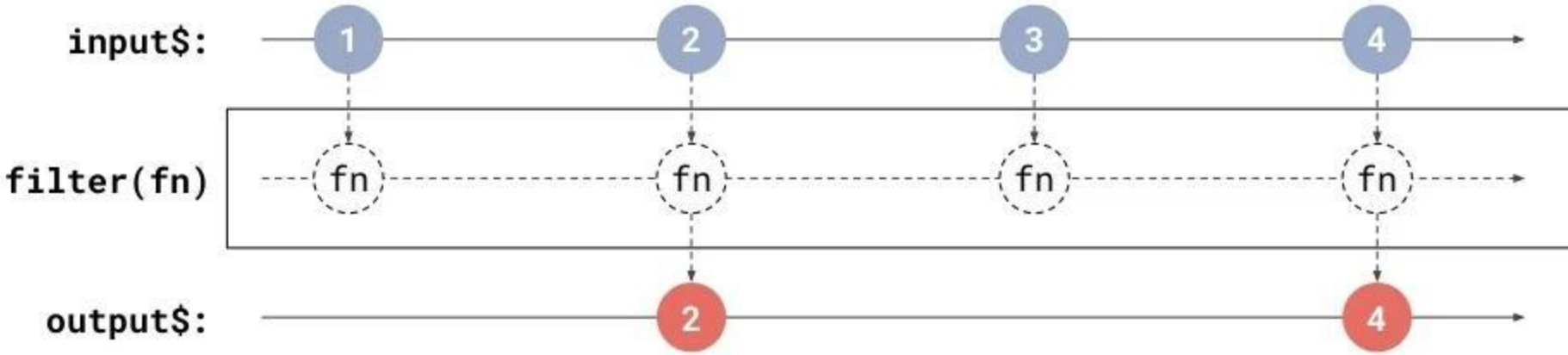
In this case it'd be better to use the native array's map, reduce, a filter methods.

https://rxjs.dev/api/operators


THE EASIEST
OPERATORS
map
filter
scan

filter

`fn: v => v % 2`



filter example

```
// RxJS v6+
//Emit only click events whose target was a DIV element


import { fromEvent } from 'rxjs';
import { filter } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const clicksOnDivs = clicks.pipe(filter(ev =>
ev.target.tagName === 'DIV'));


clicksOnDivs.subscribe(x => console.log(x));
```

filter

COMMON MISTAKES -

Using filter instead of the take or first operators.

Use first if you want to take the first value from an observable o
the first value that matches a specific criteria.

Use take if you're looking to take the first n values.

https://rxjs.dev/api/operators


THE EASIEST
OPERATORS
map
filter
scan

scan

EXPLANATION -

Applies a reducer function over the source Observable, and returns each intermediate result, with an optional seed value.

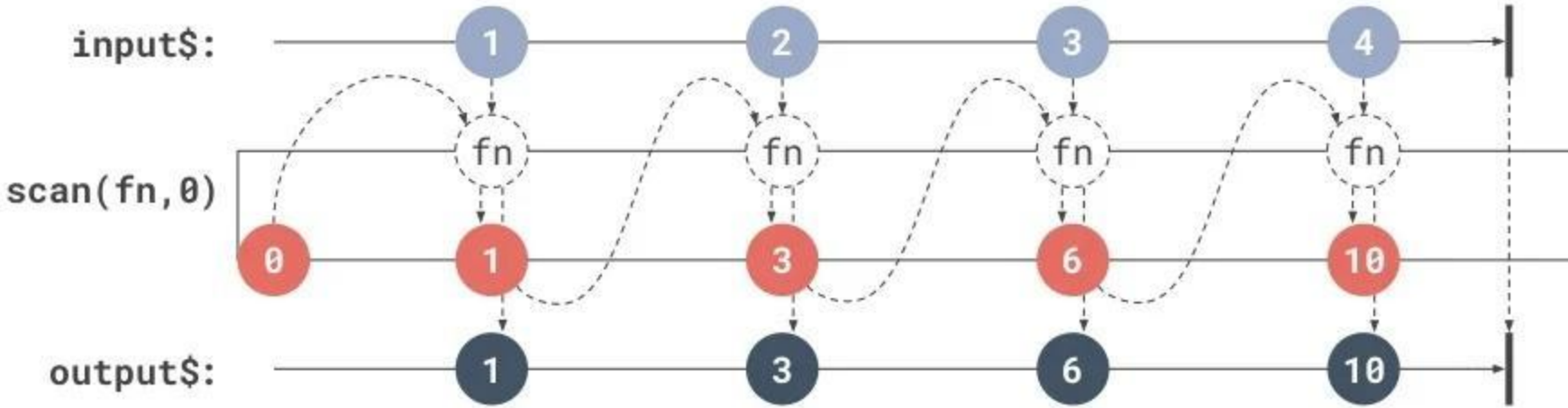
PRIMARY USES -

Managing state in a stream. No ngrx! You can use scan to create redux pattern.

https://rxjs.dev/api/operators

scan

```
fn:  (state , v) => state + v
```



scan example

```
// RxJS v6+ - Count the number of click events

import { fromEvent } from 'rxjs';
import { scan, mapTo } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const ones = clicks.pipe(mapTo(1));
const seed = 0;
const count = ones.pipe(scan((acc, one) => acc + one,
seed));
count.subscribe(x => console.log(x));
```

scan

COMMON MISTAKES -

Emitting the same reference multiple times, which can cause problems.

What you should do instead is treat your accumulated value as immutable.

https://rxjs.dev/api/operators

FLATTENING OPERATORS


FLATTENING
OPERATORS
switchMap
concatMap
mergeMap
exhaustMap


FLATTENING
OPERATORS
switchMap
concatMap
mergeMap
exhaustMap

switchMap

EXPLANATION -

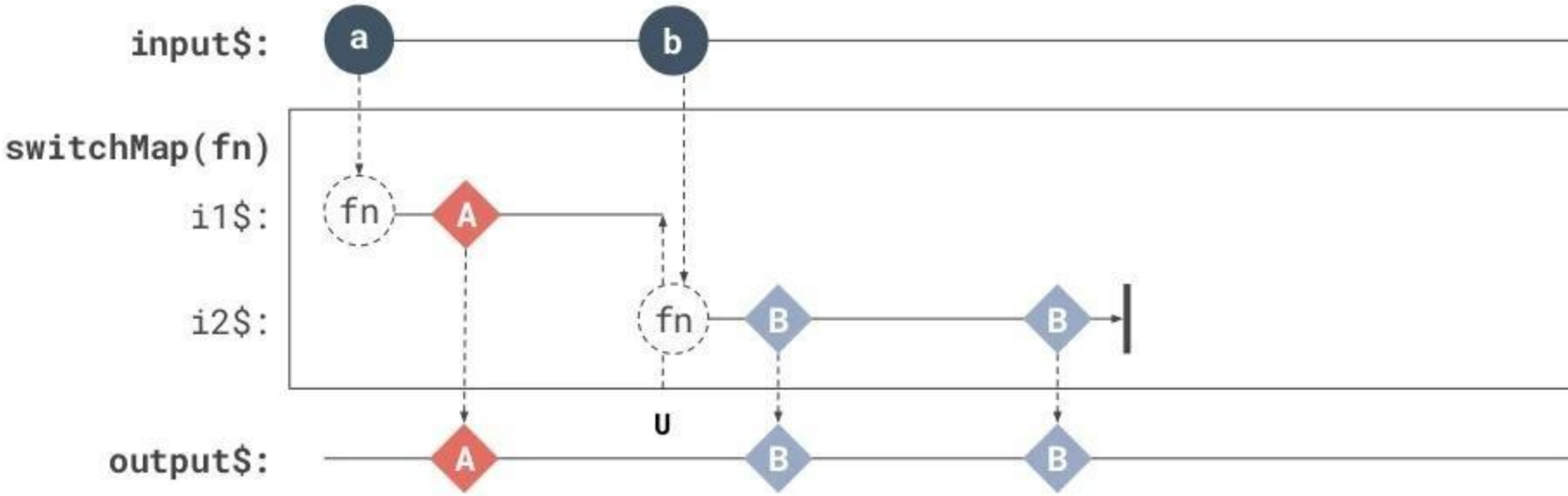
- Maps a value to a new observable,
- subscribes to that observable,
- unsubscribes from the previous observable it was subscribed to.

(switchMap only subscribes to one observable at a time)

https://rxjs.dev/api/operators/switc

switchMap


```
fn:  (v) => interval(1000).pipe(take(2))
```



switchMap example

```
// RxJS v6+ - Restart interval every 5 seconds

import { timer, interval } from 'rxjs';
import { switchMap } from 'rxjs/operators';

const source = timer(0, 5000);

const example = source.pipe(switchMap(( ) =>
interval(500)));

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

switchMap

PRIMARY USES -

Commonly used for HTTP GET.

Great for autocompletes

Toggling between two streams

Great for animation!

https://rxjs.dev/api/operators/switc

switchMap

COMMON MISTAKES -

If you're doing an HTTP POST or DELETE and a value is coming back i
the response.

If you're relying on response from POST or DELETE in order to updat
the UI, you're only going to get response from 2nd request, and not 1s

In these situations, use concatMap instead.

https://rxjs.dev/api/operators/switc


FLATTENING
OPERATORS
switchMap
concatMap
mergeMap
exhaustMap

concatMap

EXPLANATION -

Most common operator for HTTP requests.

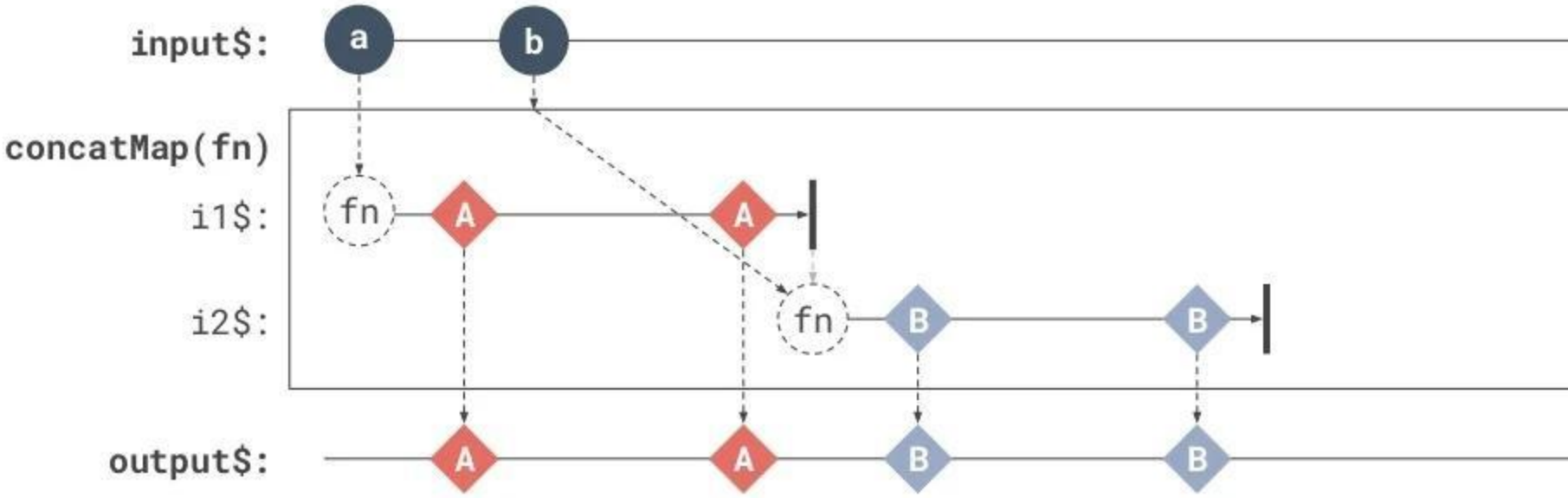
Ensures everything happens in the order it arrived.

Takes a value from the source, maps it into a new observable, an
only runs one observable at a time until it completes. Then move
on to the next value in the buffer.

https://rxjs.dev/api/operators/conc

concatMap


```
fn:  (v) => interval(1000).pipe(take(2))
```



concatMap example

```
// RxJS v6+
import { of } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { concatMap } from 'rxjs/operators';

// Emit 'ladyleet' and 'benlesh'
const source = of('ladyleet', 'benlesh');

// Map value from source into inner observable
// when complete emit result and move to next
const example = source.pipe(concatMap(username =>
ajax.getJSON(`https://api.github.com/users/${username}`));

example.subscribe(data => console.log(data));
```

concatMap

COMMON MISTAKES -

Every single time you use concatMap, you have to make sure the observa
ends.

Never use concatMap for toggling or endless streams.

If the stream never completes, every subsequent concatMap is just going
build up the buffer, and the other observables it's buffering will never run

https://rxjs.dev/api/operators/conc


FLATTENING
OPERATORS
switchMap
concatMap
mergeMap
exhaustMap

mergeMap

EXPLANATION -

Takes every single value, maps it into an observable, and subscribes to it.

Then, it outputs whatever is coming from those inner observabl as a single stream of values.

https://rxjs.dev/api/operators/merge

mergeMap

PRIMARY USES -

Sending requests to get as many responses as fast as possible when you don'
care about the response order.

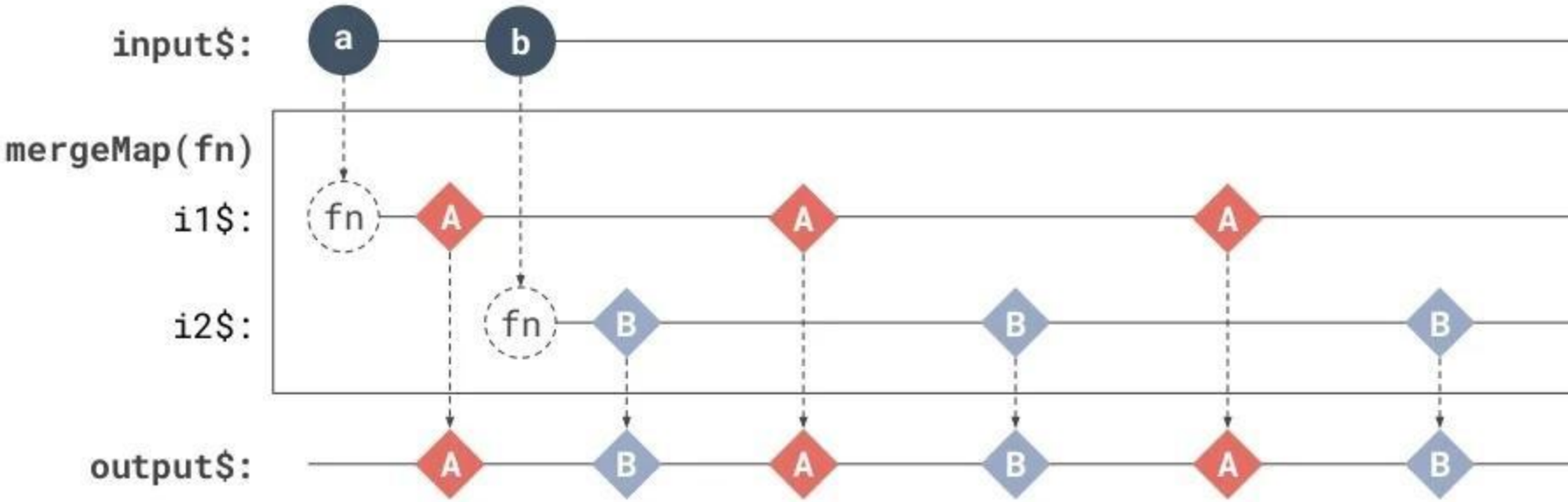
Errors from requests: mergeMap is more useful than switchMap if you want
know what errors come back from requests and be able to handle them.

(With switchMap, the subscription would be cancelled before the error com
back)

https://rxjs.dev/api/operators/merge

mergeMap

fn: (v) => interval(1000).pipe(take(2))



mergeMap example

```
// RxJS v6+
import { of } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { mergeMap } from 'rxjs/operators';

// Emit 'ladyleet' and 'benlesh'
const source = of('ladyleet', 'benlesh');

// Map value from source into inner observable
// when complete emit result and move to next
const example = source.pipe(mergeMap(username =>
ajax.getJSON(`https://api.github.com/users/${username}`));

example.subscribe(data => console.log(data));
```

mergeMap

COMMON MISTAKES -

Using mergeMap to map to HTTP requests.

You don't know how long a server will take to respond, and if one request is slow and another is fast, it will end up out of order.

(If you don't care about order, you can use it)

https://rxjs.dev/api/operators/merge


FLATTENING
OPERATORS
switchMap
concatMap
mergeMap
exhaustMap

exhaustMap

EXPLANATION - Conceptually, exhaustMap is the opposite of switchMap.

Every time a value arrives, if you're not already subscribed to a previously mapped-to observable, exhaustMap will map it into an observable and subscribe to that.

It will wait for the inner observable to complete or "exhaust" itself before it allows any more new source values to be mapped into a new inner observable.

https://rxjs.dev/api/operators/exhaustM

exhaustMap

EXAMPLE USES -

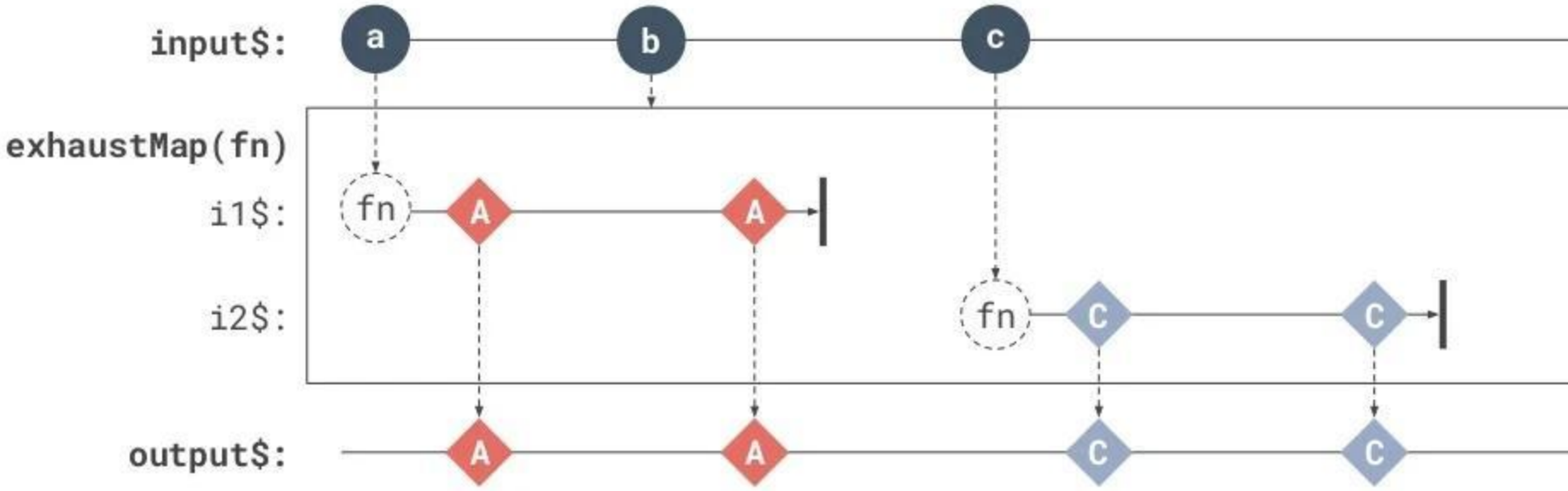
Great for preventing double submissions.

Making sure touch drag features work properly.

https://rxjs.dev/api/operators/exhaustM

exhaustMap


```
fn:  (v) => interval(1000).pipe(take(2))
```



exhaustMap example

```
// Run a finite timer for each click, only if there is no
currently active timer

import { fromEvent, interval } from 'rxjs';
import { exhaustMap, take } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const result = clicks.pipe(
  exhaustMap(ev => interval(1000).pipe(take(5)))
);
result.subscribe(x => console.log(x));
```

ERROR HANDLING


ERRORS
ERRORS EVERYWHER


ERROR
HANDLING
catchError
retry
retryWhen


ERROR
HANDLING
catchError
retry
retryWhen

catchError

EXPLANATION -

Prevents errors from traveling down the entire chain.

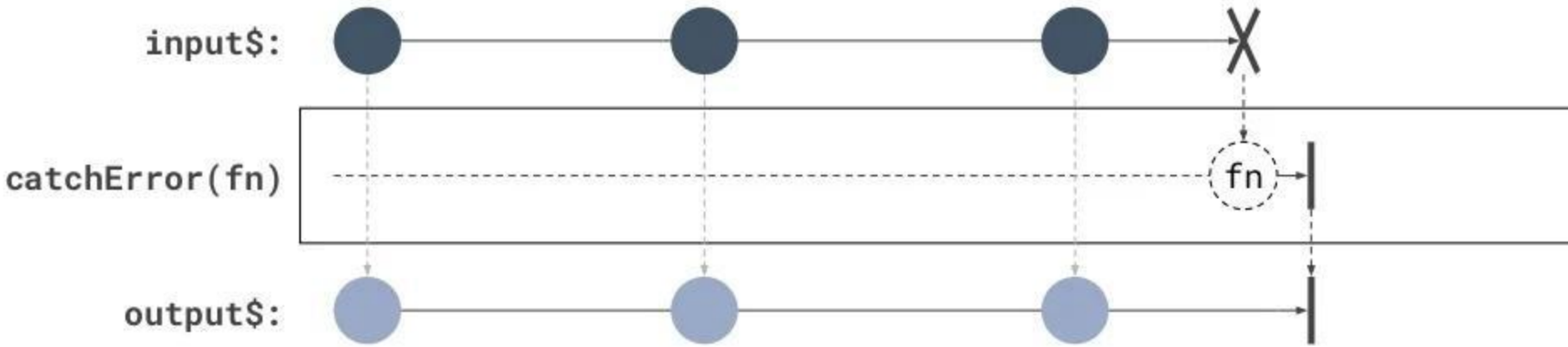
Catches errors on the Observable to be handled by returning a new Observable or throwing an error.

When it gets an error, it maps that error into a new Observable and replaces the original source observable with the new one.

https://rxjs.dev/api/operators/catchEr

catchError

`n: 3` `fn` `(e:any) => EMPTY`



catchError

PRIMARY USES -

Handling errors from observables by mapping them to new observable.

Preventing HTTP errors from killing the entire observable chain

https://rxjs.dev/api/operators/catchEr

catchError example

```
// A stream of "load button" clicks
const loadData = fromEvent(button, 'click');

// When we click "load", trigger an http get
loadData.pipe(
  concatMap(() => ajax.get('/this/will/404').pipe(
    catchError(err => {
      // side effect, to log something happened
      console.log('An error occurred loading your
stuff!');
      // Return a different observable
      return of({ message: 'We noticed an error' });
    })
 ))
).subscribe(x => console.log(x));
```


ERROR
HANDLING
catchError
retry
retryWhen

retry

EXPLANATION -

If your observable errors, it will resubscribe to that observable a
specified number of times.

https://rxjs.dev/api/operators/

retry

PRIMARY USES -

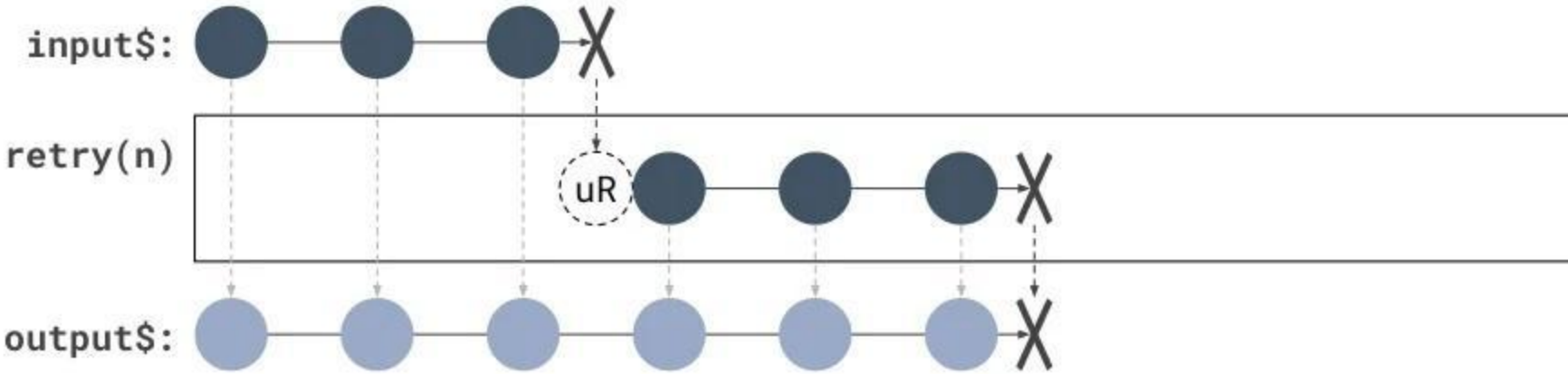
Great in the case of an HTTP request or websocket stream whe
there is network flakiness.

When it gets an error signal, it resubscribes to the original sourc

https://rxjs.dev/api/operators/

retry

n:1 (uR) Unsubscribes and recycles observable



retry example

```
// RxJS v6+ - Retry 2 times on error
import { interval, of, throwError } from
'rxjs';
import { mergeMap, retry } from
'rxjs/operators';

const source = interval(1000);
const example = source.pipe(
  mergeMap(val => {
    //throw error for demonstration
    if (val > 5) {
      return throwError('Error!');
    }
    return of(val);
  }),
  //retry 2 times on error
  retry(2)
);

const subscribe = example.subscribe({
  next: val => console.log(val),
  error: val => console.log(`${val}:
Retried 2 times then quit!`)
});
```

retry

COMMON MISTAKES -

Using retry without an argument.

Using retry on hot observables.

(If you're using a Subject or stateful observable, it's not going to reset the state of that observable, just resubscribe to it)

https://rxjs.dev/api/operators/


ERROR
HANDLING
catchError
retry
retryWhen

retryWhen

EXPLANATION -

Gives you more control than retry.

Gives you an observable of errors that you can map into an observable of notifications of when you want to retry.

https://rxjs.dev/api/operators/retryW

retryWhen

PRIMARY USES -

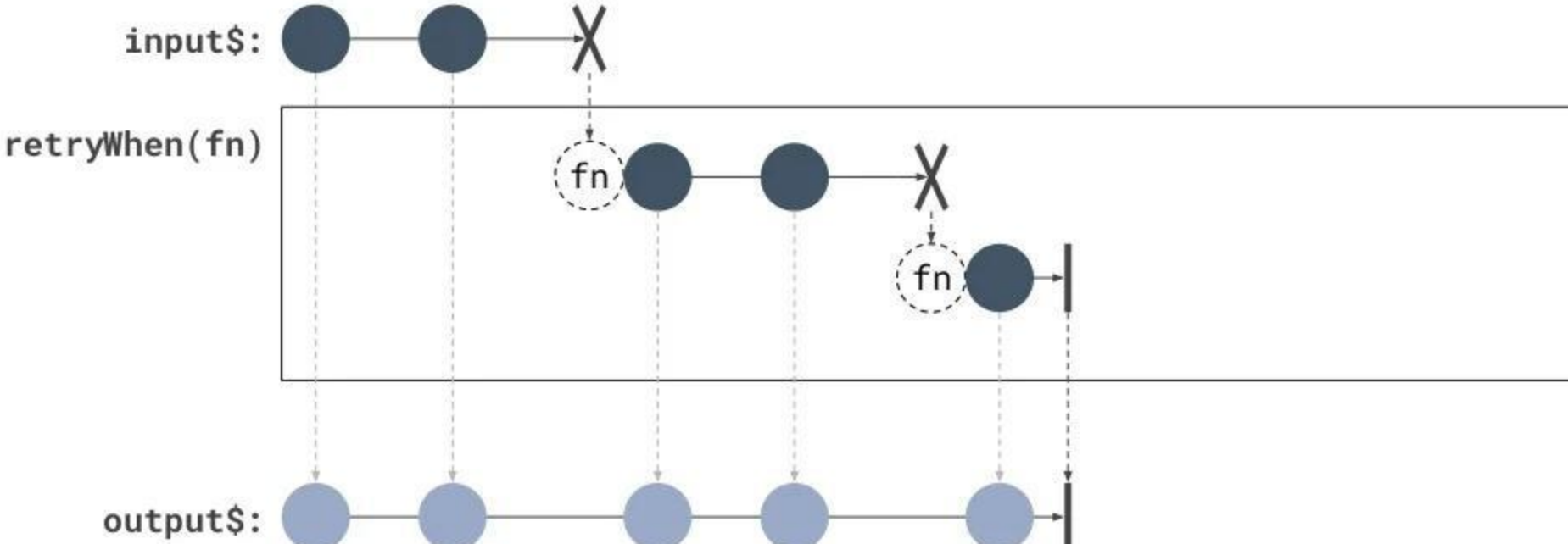
Using retry with a delay i.e incremental step back on network failure

(if you send a network request and it fails - wait 2 seconds, try again, if it fails, wait 4 and try, then 8 and try, etc.)

https://rxjs.dev/api/operators/retryW

retryWhen

n: 1 (fn) (errors$) => errors$.pipe(switchMap(e => e.message === 'X' ? throwError(e) : of(1)))



retryWhen example

```
// RxJS v6+ - Trigger retry after specified duration
import { timer, interval } from 'rxjs';
import { map, tap, retryWhen, delayWhen } from 'rxjs/operators';

//emit value every 1s
const source = interval(1000);
const example = source.pipe(
  map(val => {
    if (val > 5) {
      //error will be picked up by retryWhen
      throw val;
    }
    return val;
  })
```

```
retryWhen(errors =>
    errors.pipe(
      //log error message
      tap(val => console.log(`Value ${val} was too high!`)),
      //restart in 6 seconds
      delayWhen(val => timer(val * 1000))
    )));

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

COMPLETION OPERATORS


COMPLETION
OPERATORS
take
takeUntil
takeWhile
first


COMPLETION
OPERATORS
take
takeUntil
takeWhile
first

take

EXPLANATION -

Takes the first specified number of values.

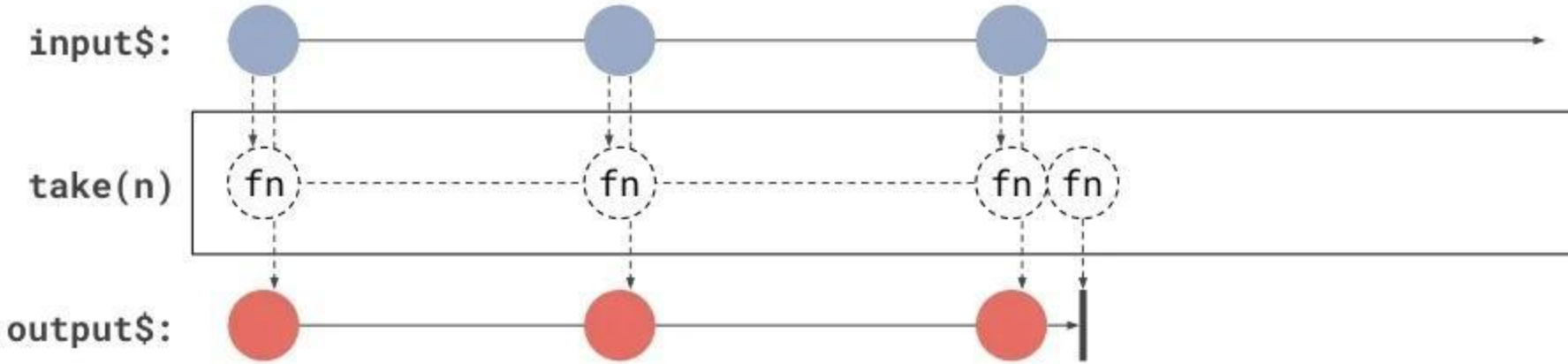
PRIMARY USES -

If you want to take the first n values from the source observable

https://rxjs.dev/api/operators

take

`n: 3` `fn` `(v) => v < n`



take example

```
import { interval } from 'rxjs';

import { take } from 'rxjs/operators';


const intervalCount = interval(1000);

const takeFive = intervalCount.pipe(take(5));


takeFive.subscribe(x => console.log(x));
```


COMPLETION
OPERATORS
take
takeUntil
takeWhile
first

takeUntil

EXPLANATION -

Using a notifier observable to tell a source observable to complete.

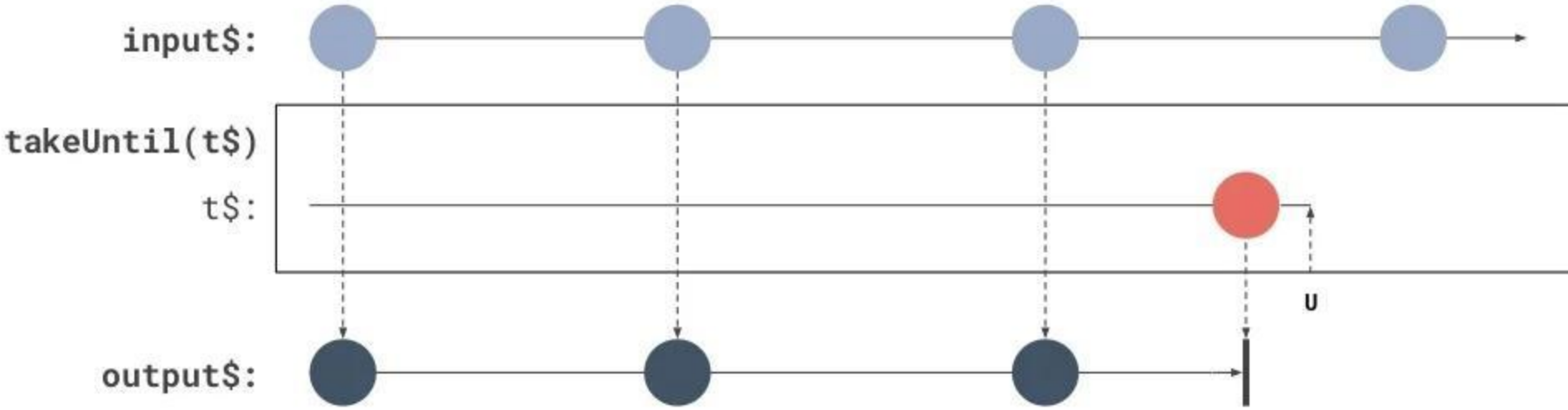
PRIMARY USES -

Declarative subscription management

https://rxjs.dev/api/operators/takeU

takeUntil

```
t$:  fromEvent(document.body, 'click')
```



takeUntil example

```
import { fromEvent, interval } from 'rxjs';
import { takeUntil } from 'rxjs/operators';

const source = interval(1000);

const clicks = fromEvent(document, 'click');

const result = source.pipe(takeUntil(clicks));

result.subscribe(x => console.log(x));
```

takeUntil

COMMON MISTAKES -

Providing the source observable with a notifier that never emits
completes.

https://rxjs.dev/api/operators/take


COMPLETION
OPERATORS
take
takeUntil
takeWhile
first

takeWhile

EXPLANATION -

Emits values emitted by the source observable so long as each value satisfies the given predicate…

and then completes as soon as this predicate is not satisfied.

https://rxjs.dev/api/operators/takeW

takeWhile example

```
import { fromEvent } from 'rxjs';
import { takeWhile } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const result = clicks.pipe(takeWhile(ev => ev.clientX > 200));
result.subscribe(x => console.log(x));
```

takeWhile's new feature (inclusive)

```
import { fromEvent } from 'rxjs';
import { takeWhile } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const result = clicks.pipe(takeWhile(ev => ev.clientX >
200, true));
result.subscribe(x => console.log(x));
```

takeWhile

PRIMARY USES -

Limit for mouse movements
Progress bars

COMMON MISTAKES - Not realizing that takeWhile has to wait for a value to arrive before it completes and using it when you actually wanted the observable to complete after an external event.

In this scenario you should have used takeUntil.

https://rxjs.dev/api/operators/takeW


COMPLETION
OPERATORS
take
takeUntil
takeWhile
first

first

EXPLANATION - Emits only the first value (or the first value that mee
some condition) emitted by the source Observable, or a default value

If no default value is provided, it will error if it does not get a first valu

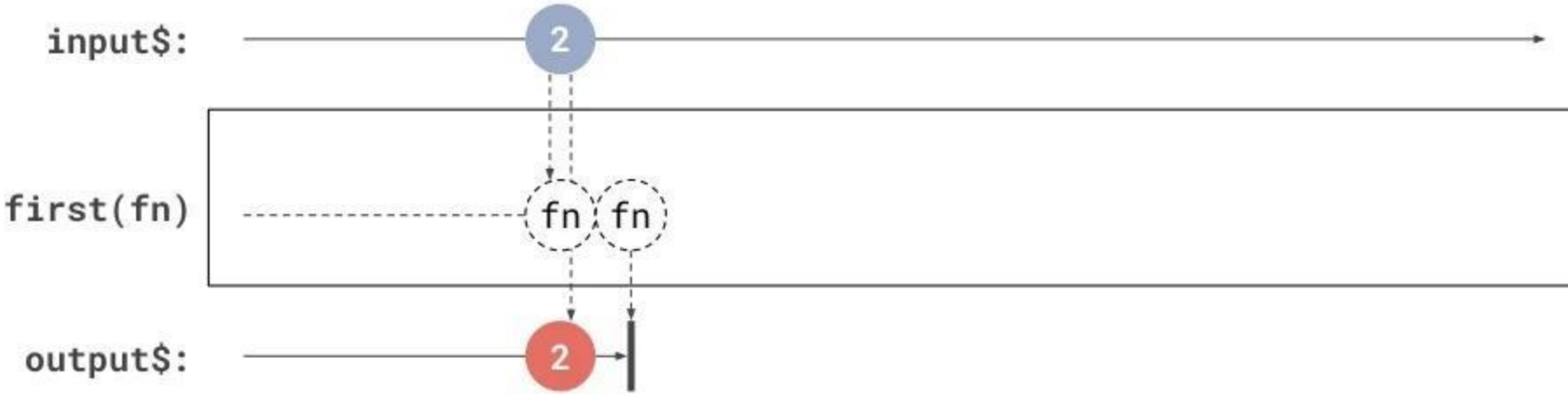
PRIMARY USES -

Used when you want to get the very next value from a websocket or
from an update interval.

https://rxjs.dev/api/operators

first

fn (v) => v < 3



first example

```
import { fromEvent } from 'rxjs';
import { first } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const result = clicks.pipe(first(ev => ev.target.tagName
=== 'DIV'));

result.subscribe(x => console.log(x));
```

first

COMMON MISTAKES -

Not realizing that first will error if it never finds a first value, and does not have a default value.

(In this case, you probably wanted a filter and a take)

https://rxjs.dev/api/operators

SUBJECTS AND MULTICASTING

SUBJECTS AND MULTICASTING

- Subject
- BehaviorSubject
- ReplaySubject
- share
- shareReplay

SUBJECTS AND MULTICASTING

- Subject
- BehaviorSubject
- ReplaySubject
- share
- shareReplay

Subject

EXPLANATION -

A Subject is an observable that is also an observer.

PRIMARY USES -

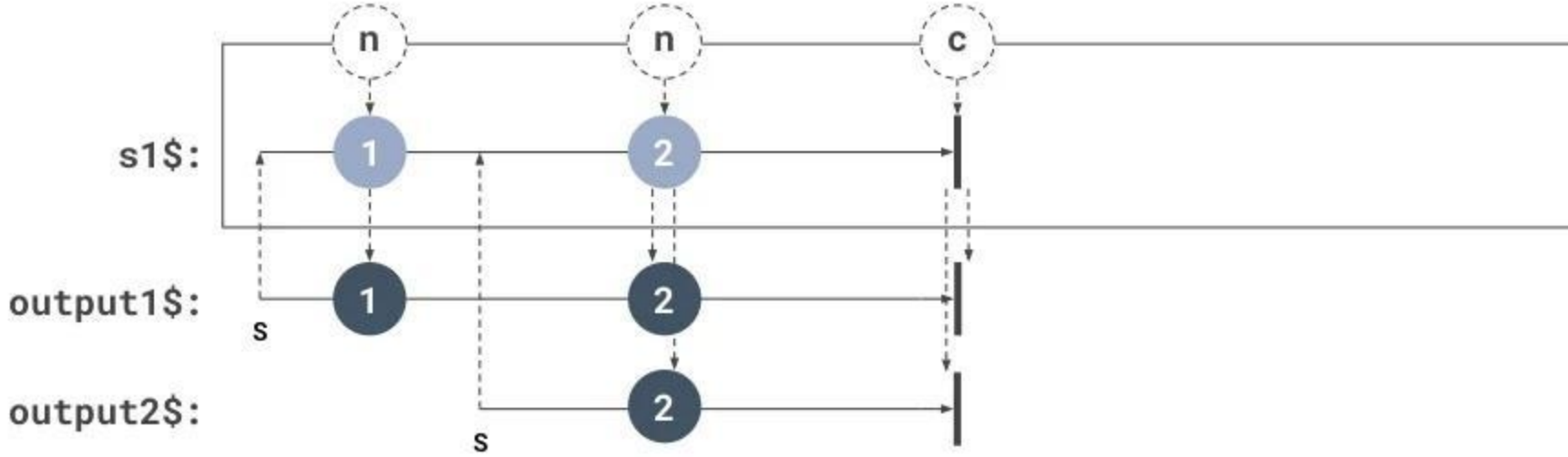
Multicasting is the main use case for Subject.

Getting an observable out of user events ie button clicks.

https://rxjs.dev/api/index/class/Sul

Subject

s1$: new Subject() (n) next(v) (c) complete()



Subject

COMMON MISTAKES - You should not just use Subject because you don't understand how to create an observable.

With Subjects, you do not get guaranteed teardown and memor management that you would get with Observable.

After a Subject has errored or completed, it's no longer usable.

https://rxjs.dev/api/index/class/Su

SUBJECTS AND MULTICASTING

- Subject
- BehaviorSubject
- ReplaySubject
- share
- shareReplay

BehaviorSubject

EXPLANATION -

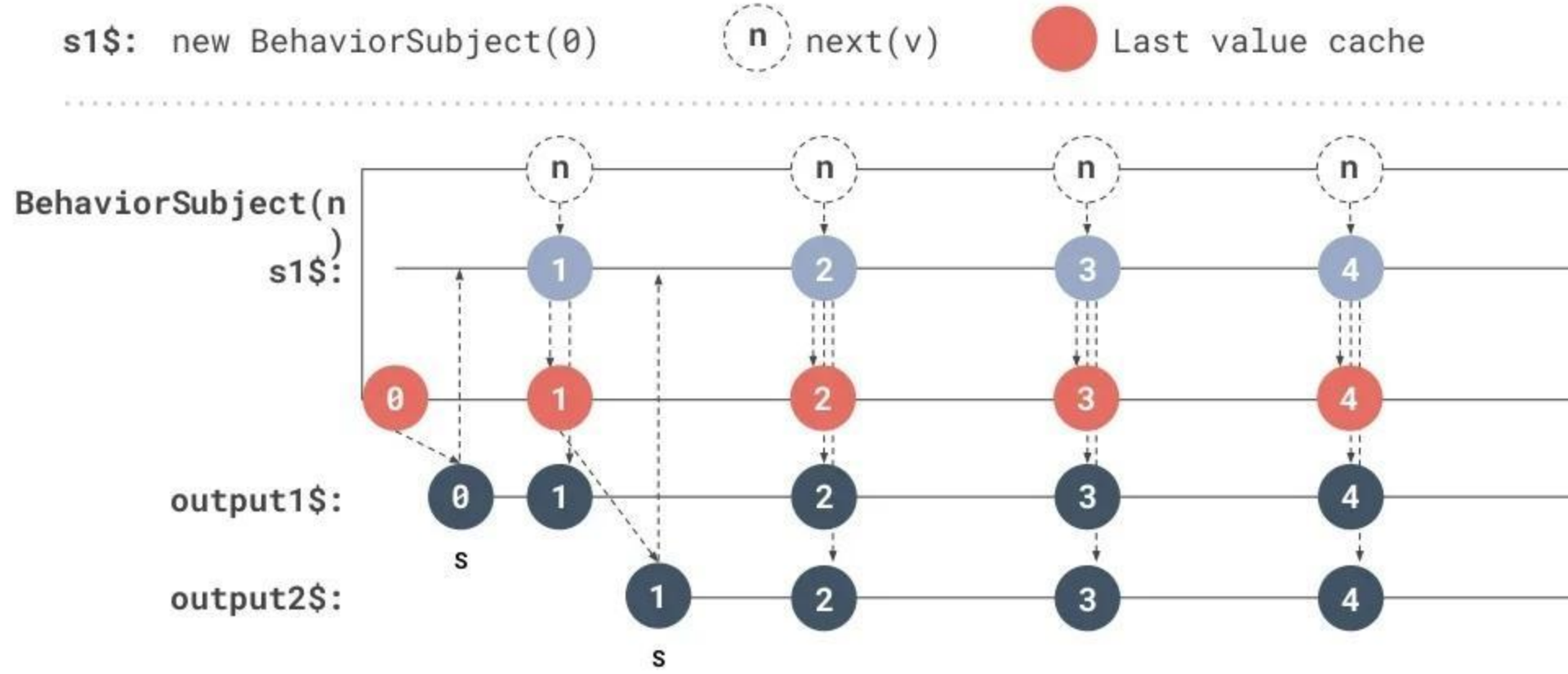
A variant of Subject that requires an initial value and emits its current value whenever it is subscribed to.

PRIMARY USES -

When you don’t want a user or system to wait to get the next value.

https://rxjs.dev/api/index/class/BehaviorSu

BehaviorSubject



SUBJECTS AND MULTICASTING

- Subject
- BehaviorSubject
- ReplaySubject
- share
- shareReplay

ReplaySubject

DOCS EXPLANATION - Variant of Subject that "replays" old values to new subscribers.

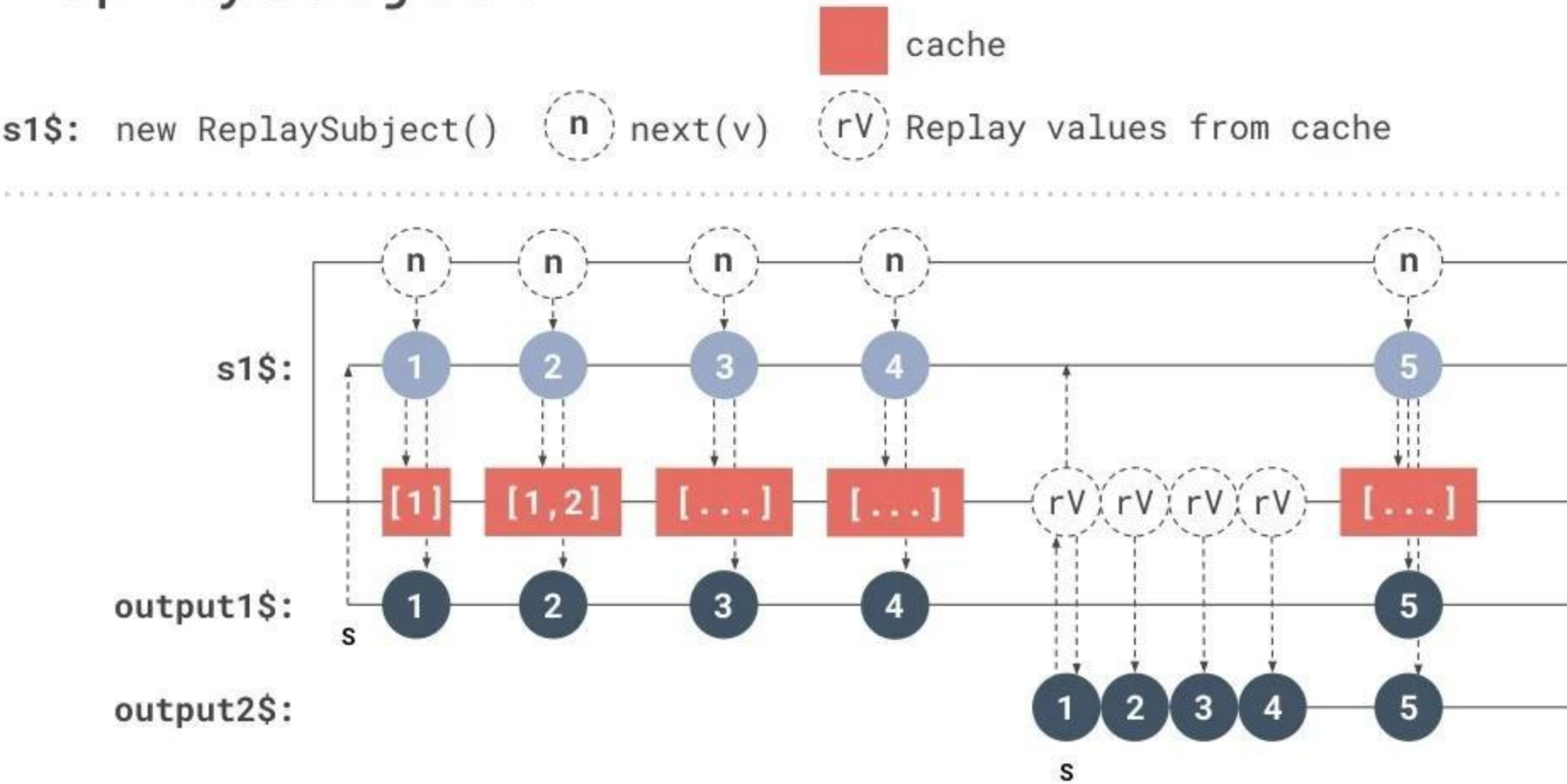
Buffers a specified count of values for a specified amount of time and emits those values immediately to any new subscribers (in addition to emitting new values to existing subscribers).

PRIMARY USES - What shareReplay uses under the hood. Typically do not want to use this raw.

Use share or shareReplay instead.

https://rxjs.dev/api/index/class/ReplaySub

ReplaySubject



SUBJECTS AND MULTICASTING

- Subject
- BehaviorSubject
- ReplaySubject
- share
- shareReplay

share

EXPLANATION - Returns a new Observable that multicasts (shares) the original Observable.

As long as there is at least one Subscriber this Observable will b subscribed and emitting data.

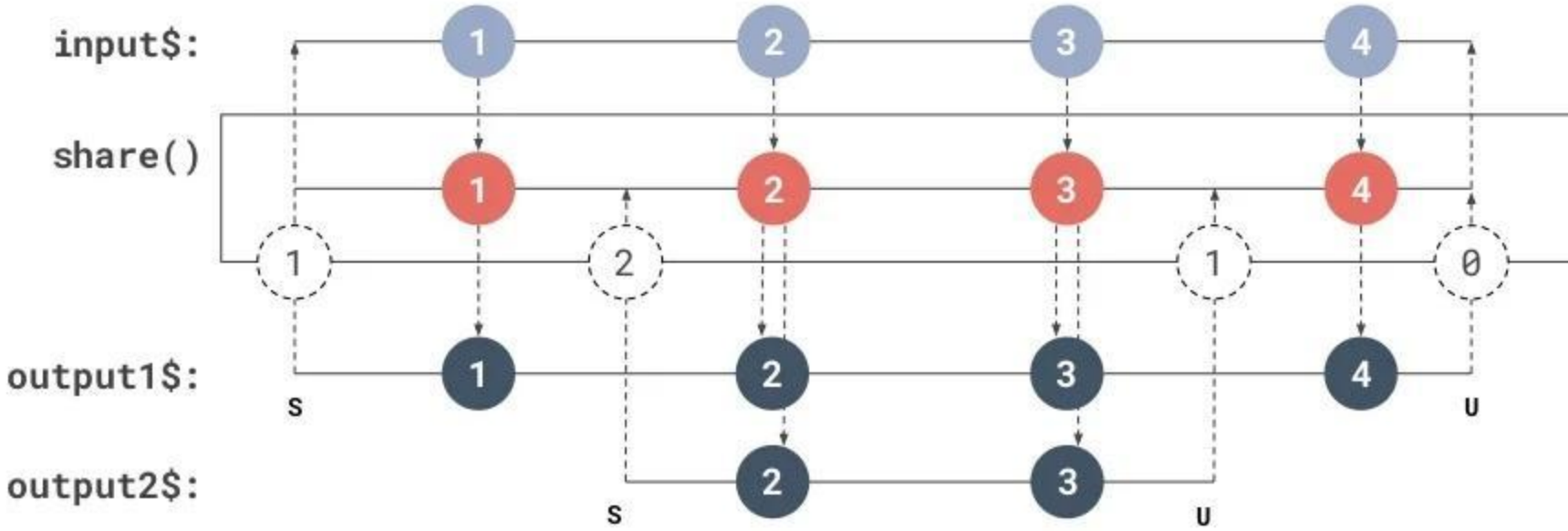
When all subscribers have unsubscribed it will unsubscribe from the source Observable.

PRIMARY USES - Multicasting

https://rxjs.dev/api/operators

share (simple)

input$: interval(1000) (0) Number of subscriptions



share

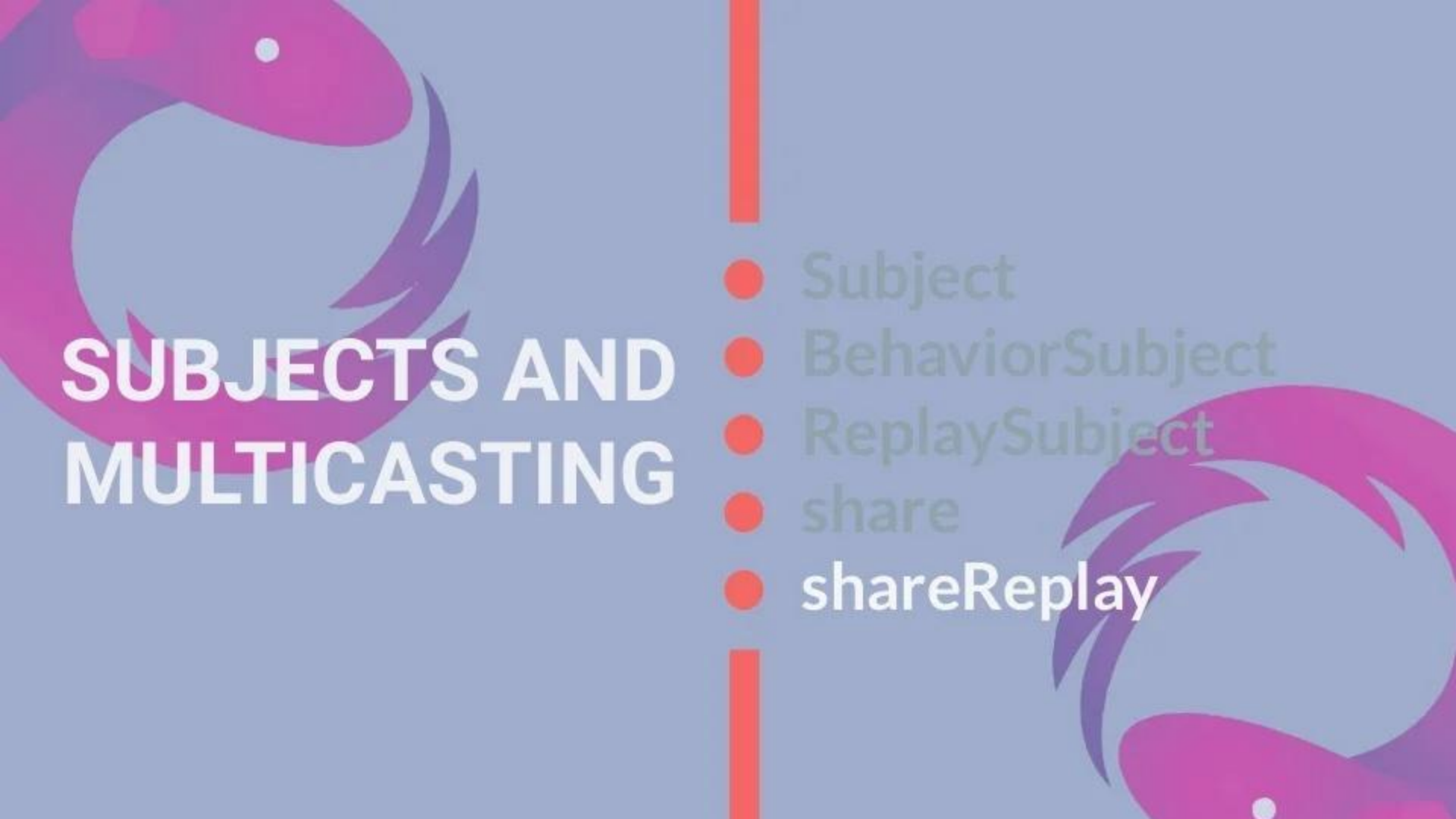
COMMON MISTAKES - Using publish or multicast when you should be using share or shareReplay.

The reason you should use share / shareReplay is because under the hood publish and multicast only have one instance of subject, so you can't retry it i there is an error.

Share automatically recycles the subject under the hood to allow you to resubscribe to the subject.

When it errors or completes it will recycle and create a brand new instance s it's not completely dead.

https://rxjs.dev/api/operators


SUBJECTS AND MULTICASTING
Subject
BehaviorSubject
ReplaySubject
share
shareReplay

shareReplay

EXPLANATION - Same as share, but keeps the underlying ReplaySubject so next subscriber gets all the successful results and successful completion.

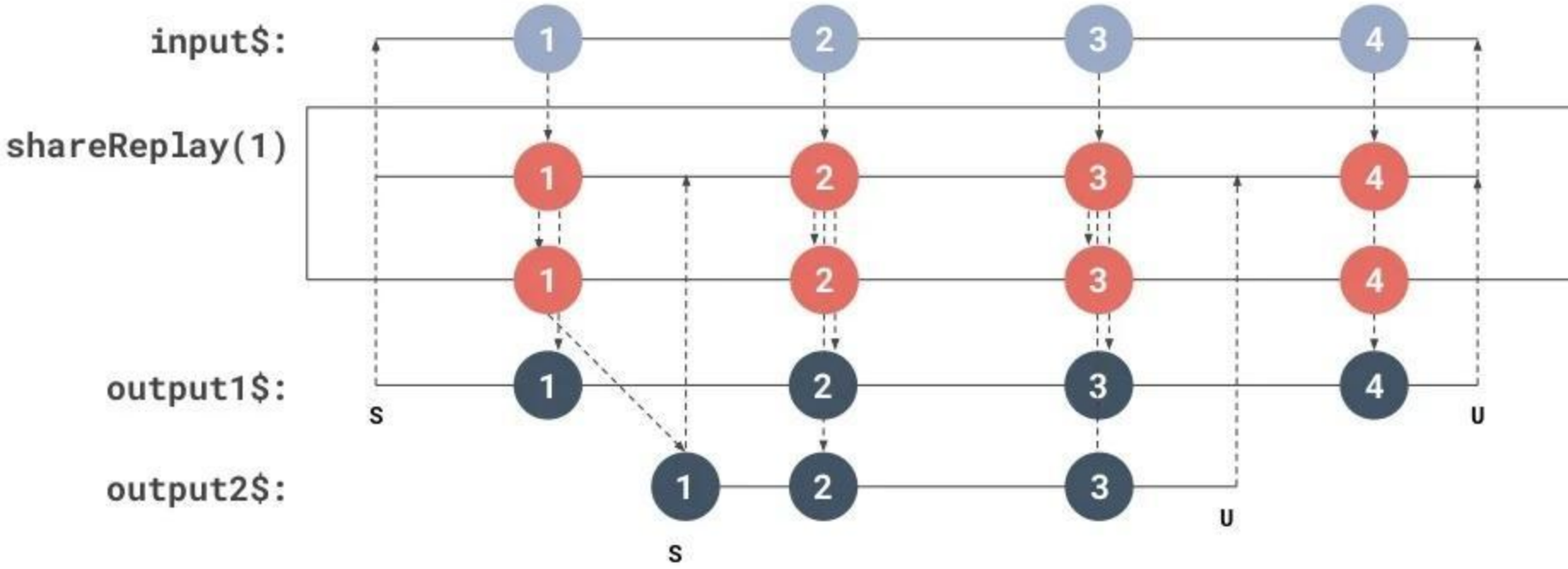
shareReplay does ref counting under the hood, but keeps a ref count of 1 to make sure it can run to completion and to ensure the next subscriber gets th last cached value.

PRIMARY USES - Retrieving expensive data via HTTP and you don't want to it twice

https://rxjs.dev/api/operators/share

shareReplay (simple)

input$: interval(1000)



shareReplay

COMMON MISTAKES -

There is a configuration object - after the last subscriber unsubscribes you can actually recycle, but that is not the default behavior and it confuses a lot of people.

You should always have at least 1 argument passed into ShareReplay.

ShareReplay without an argument will cache every single value that it had nexted into it and replay those to new subscribers.

https://rxjs.dev/api/operators/share

There are 60+ operators in RxJS

- Flattening Operators
- Error Handling
- Completion Operators
- Subjects & Multicasting

RxJS OPERATOR VIDEO SERIES

RxJS Operators - Real World Use Cases with Be

Yolo Brolo - 2 / 5

1 

RxJS Operators - Explanation, Real World Use Cases, and Anti
Yolo Brolo

▶ 

RxJS Operators part 2 - Explanation, Real World Use
Yolo Brolo

3 

Operators in RxJS - Recursion ... and expand.
Yolo Brolo

4 

RXJS Subjects - Real World Use Cases - Async Subject,
Yolo Brolo

5 

RxJS - Real World Operators - Scan, Completion Operators &
Yolo Brolo

https://bit.ly/2IstZwM

COME CONTRIBUTE!

THANK YOU

@benlesh
Angular Core Team,
RxJS Author

@michael_hladky
Google Developer
Expert, Angular

OUR LINKS

/thisdotmedia

This Dot Media

www.thisdot.co

news.thisdot.co

THANK YOU!

hi@thisdot.co