RxJS 5 Workshop

Ben Lesh

@benlesh

What is RxJS?

“lodash for events”

What is RxJS?

“lodash for async”

Types of Async in Web Apps

- AJAX
- User events (mouse, keyboard, touch, etc)
- Web sockets
- Workers
- Animations
- SSE

Methods for dealing with async

- callbacks
- promises
- observables
- generators, CSP and others

Callbacks

```
getSomeData((data) => {
  doSomething(data);
});
```

Callbacks

```
getSomeData((err, data) => {
  if (err) {
    handleError(err);
  } else {
    doSomething(data);
  }
});
```

Callback Hell

```
getSomeData((data) => {
    foo(data);
    getSomeData((data) => {
        bar(data);
        getSomeData((data) => {
            baz(data);
        });
    });
});
```

Promises

```
getSomeData()
    .then((data) => {
        foo(data);
        return getSomeData();
    })
    .then((data) => {
        bar(data);
        return getSomeData();
    })
    .then((data) => {
        baz(data);
    });
```

Promises

- Guaranteed future
- Single value
- Immutable
- Multicast (caching)
- Eager (not lazy)

These two features can be a problem

A “Promise” to a future value

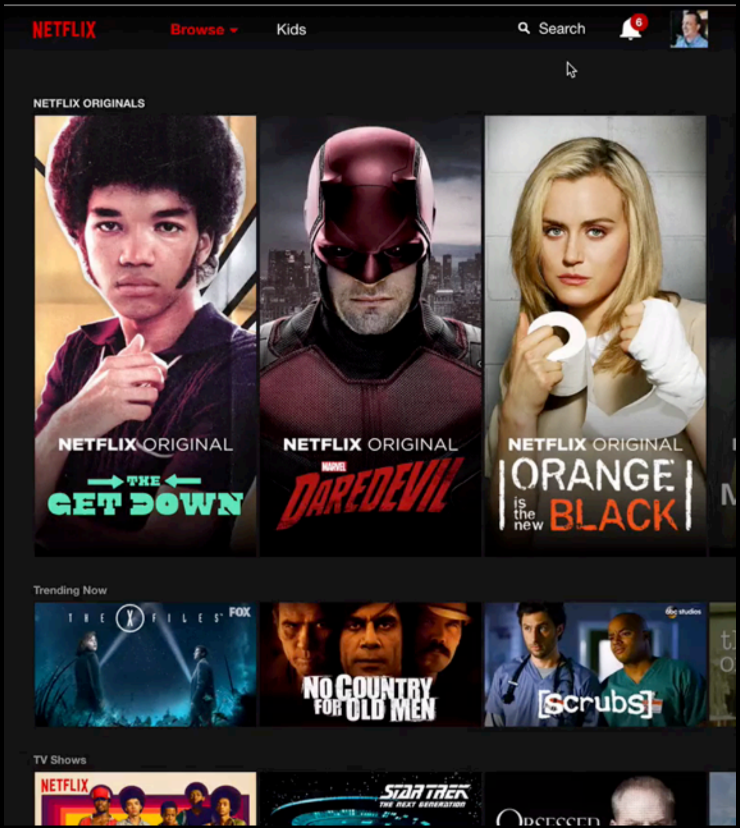
Promises can’t be cancelled

Cancellation

- Prevents code from being called unnecessarily
- Calls tear down logic

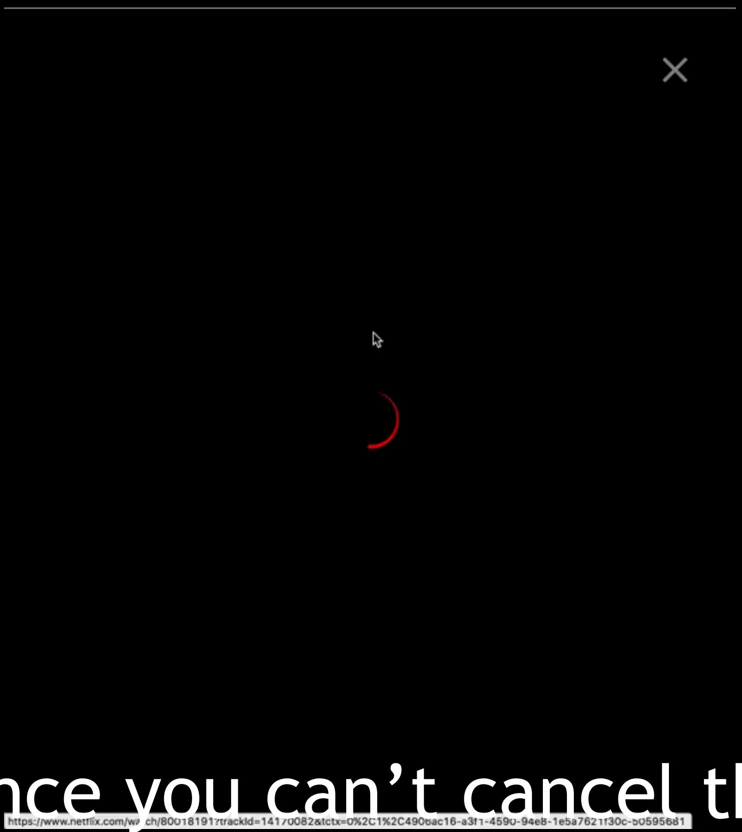
Loading view data without cancellation

Loading view data without cancellation

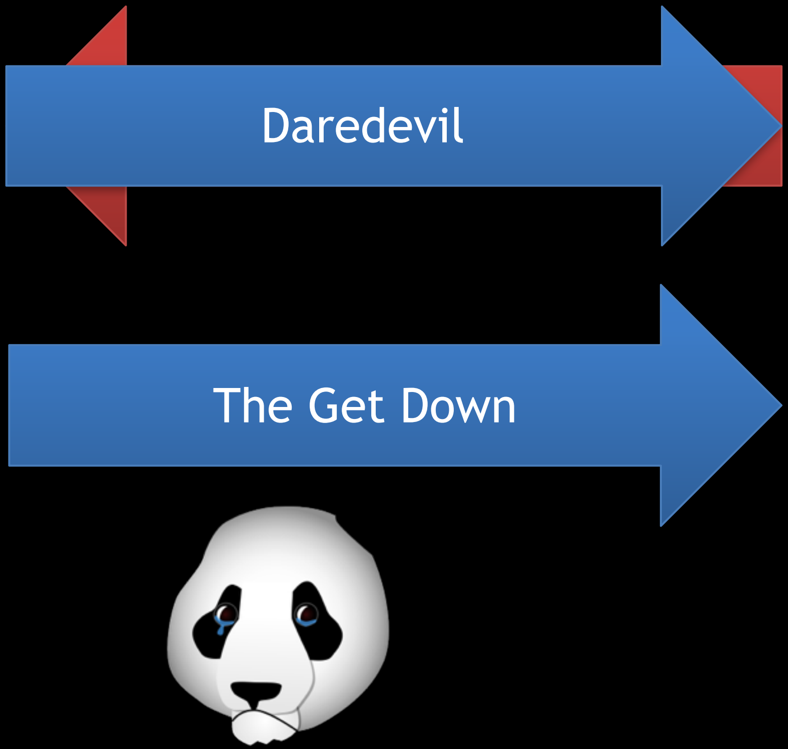


Daredevil

Loading view data without cancellation

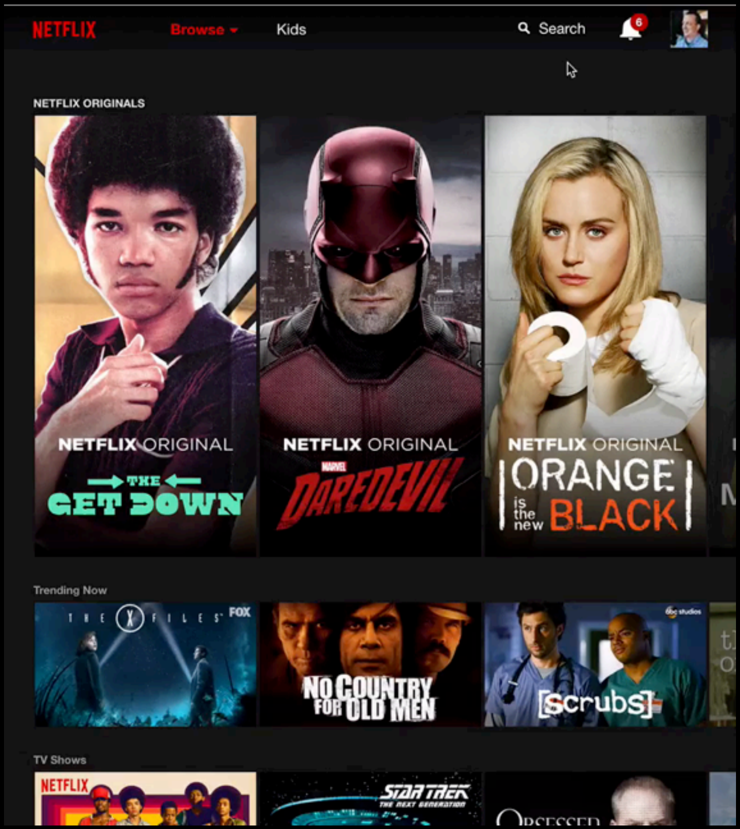


https://www.netflix.com/watch/80018191?trackId=14170082&tctx=0%2C1%2C490bac16-a3f1-4590-94e8-1e5a7621f30c-50595681



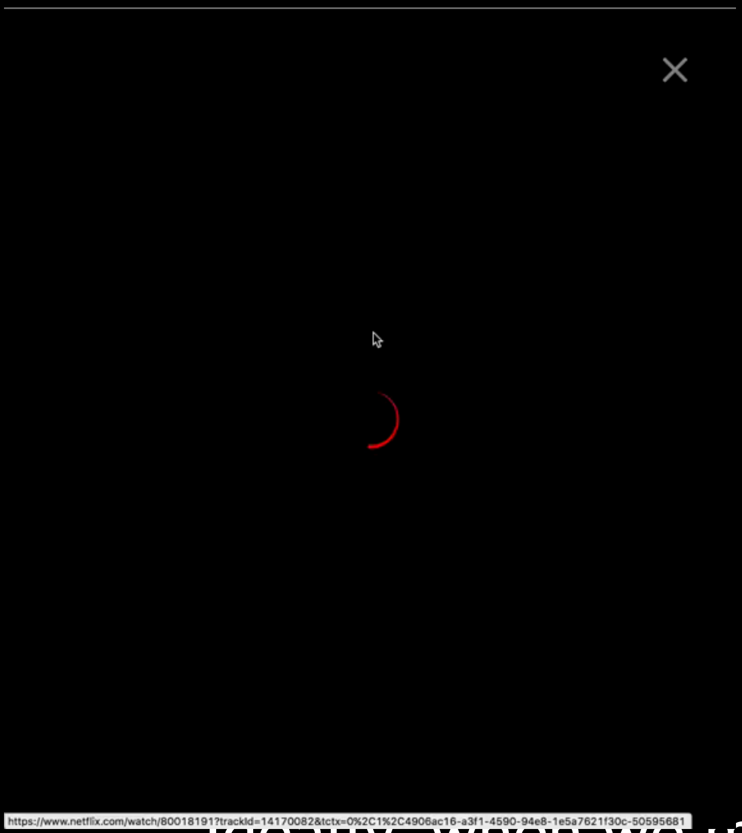
Since you can’t cancel the previous promise,
you’re stuck processing the response
and somehow signaling “disinterest”

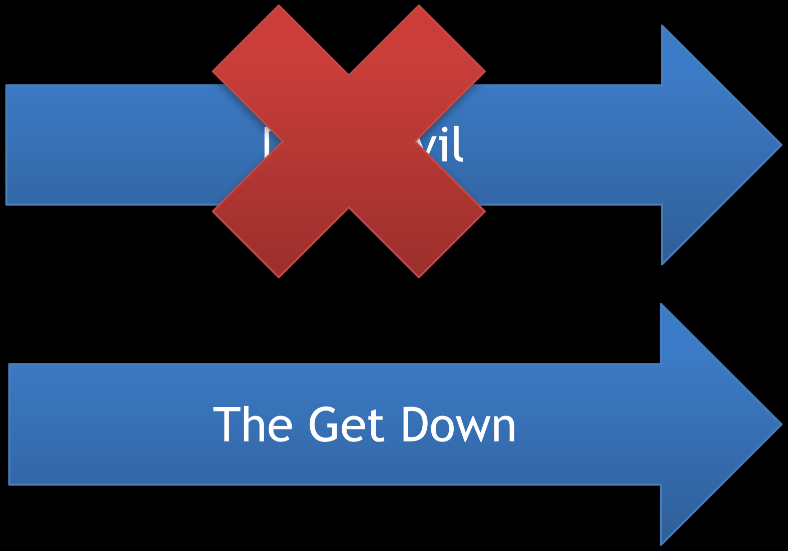
A better scenario



Daredevil

A better scenario


https://www.netflix.com/watch/80018191?trackId=14170082&tctx=0%2C1%2C4906ac16-a3f1-4590-94e8-1e5a7621f30c-50595681



Ideally, when we make a new request
we can abort the the old one
so it’s never handled and processed

We want an async type with cancellation

Promises are a single value

- AJAX
- User Events (clicks, mousemoves, keyups, etc)
- Animations
- Sockets
- Workers

We want a type that can handle more than one value

JavaScript has a type for more than one value

Iterable

Iterable

- iterable.iterator() to get an iterator
- iterator.next() to get a result
    - result.value: the value yeilded
    - result.done: whether or not it’s complete
- errors are thrown during iterator.next() call

Iterable

```
const iterator = iterable.iterator();

while (true) {
  try {
    let result = iterator.next();
  } catch (err) {
    handleError(err);
  }
  if (result.done) {
    break;
  }
  doSomething(result.value);
}
```

Iterables alone aren’t great for async

- Poll it?
  - All sorts of problems
- Iterator of Promise: Have each value in your result be a promise.
  - Good for backpressure
  - But it allocates a promise for each value (imagine mouse move events)
  - Not great for events where you don’t always need push/pull (e.g. WebSockets)

Observable

Iterable turned inside out

Iterator -> Observer

Instead a method to get a value

```
let result = iterator.next();
let value = result.value;
// do stuff
```

We have a method that accepts values

```
observer.next = (value) => { /* do stuff */ };
```

Iterator -> Observer

Instead of throwing errors when we call next()

```
try {
  let result = iterator.next();
} catch (err) {
  // handle error
}
```

We push errors to a method as they happen

```
observer.error = (err) => { /* handle error */ };
```

Iterator -> Observer

Instead of needing to check `done` on the result for completion

```
let result = iterator.next();
if (result.done) {
  // handle completion
}
```

We push completions to a method as they happen

```
observer.complete = () => { /* handle error */ };
```

Observer

```
const observer = {
  next(value) { /* handle value */ },
  error(err) { /* handle error */ },
  complete() { /* handle complete */ }
};
```

Iterable -> Observable

Instead of a method that returns an iterator

```
let iterator = iterable.iterator();
```

We have a method that accepts an observer

```
observable.observer(observer);
observable.subscribe(observer);
```

Observable is the “dual” of Iterable

It allows us to push values over time

What about cancellation?

That’s easier to show you if we look at how Observables are created

Use the Observable constructor

```
var myObservable = new Rx.Observable();
```

Pass it a subscriber function that gives you an observer

```
var myObservable = new Rx.Observable((observer) =>
{});
```

Use `next` on the observer to emit values from your observable

```
var myObservable = new Rx.Observable((observer) => {
  observer.next(‘hello world!’);
});
```

Call `complete` to signal the observable is done successfully

```
var myObservable = new Rx.Observable((observer) => {
  observer.next(‘hello world!’);
  observer.complete();
});
```

or use `error` to signal a problem caused the observable to stop

```
var myObservable = new Rx.Observable((observer) => {
  observer.next(‘hello world!’);
  observer.error(new Error(‘sad things’));
});
```

Observables are lazy!

Remember: They won’t do anything until you subscribe!

Subscribe using the `subscribe` function

```
myObservable.subscribe();
```

Provide `subscribe` with an observer

```
myObservable.subscribe({
  next: x => console.log(‘next’, x),
  error: err => console.error(err),
  complete: () => console.info(‘done’)
});
```

As shorthand, RxJS `subscribe` can also take up to 3 handlers

```
myObservable.subscribe();
```

As shorthand, RxJS `subscribe` can also take up to 3 handlers

```
myObservable.subscribe(
  x => console.log(‘next’, x) // next
);
```

As shorthand, RxJS `subscribe` can also take up to 3 handlers

```
myObservable.subscribe(
  x => console.log(‘next’, x),
  err => console.error(err) // error
);
```

As shorthand, RxJS `subscribe` can also take up to 3 handlers

```
myObservable.subscribe(
  x => console.log(‘next’, x),
  err => console.error(err),
  () => console.info(‘done’) // complete
);
```

Try it out

examples/node/easy-as-123.js

Recap: Observables

- any number of values
- any amount of time
- lazy
- cancellable
- “sets” like iterables
- push values

Sync vs Async

… but this is an async type???

Beware: Sync vs Async

```
var myObservable = new Rx.Observable((observer) => {
  observer.next(‘hello world!’);
  observer.complete();
});

console.log(‘before subscribe’);

var subscription = myObservable.subscribe(
  x => console.log(‘next’, x),
  err => console.error(err),
  () => console.info(‘done’)
);

console.log(‘after subscribe’);
```

Synchronous??

```
> “before subscribe”
> “next” “hello world”
> “done”
> “after subscribe”
```

Why allow synchronous behavior?

- DOM events can be registered and triggered in the same job.
- Observables are just functions…

Async behavior is determined by producer

```
var myObservable = new Rx.Observable((observer) => {
  var id = setTimeout(() => {
    observer.next(‘hello world!’);
    observer.complete();
  });
  return () => clearTimeout(id);
});

console.log(‘before subscribe’);

var subscription = myObservable.subscribe(
  x => console.log(‘next’, x),
  err => console.error(err),
  () => console.info(‘done’)
);

console.log(‘after subscribe’);
```

TRY IT

examples/node/sync-vs-async.js

Part 1a – Observable Creators

Common Observable Types

- Observable.of(a) – scalar value
- Observable.of(a, b, c) – sync array
- Observable.empty() – just completes
- Observable.never() – never emits or completes
- Observable.throw(new Error()) – sync throw

Scalar Observables

Observables of a single, synchronous value.

```
Observable.of(1);
Observable.from([‘one’]);
```

In RxJS 5 there are performance optimizations for scalar observables.

Empty Observables

Never emits, just completes.

```
Observable.empty()
```

- Always returns the same, static instance
- Used as “null” in merges
- Used to complete `retryWhen`
- Optimizations in RxJS 5

Error Observables

Just throws immediately

Observable.throw()

Observable.throw(new Error(‘test’))

Roughly equivalent to Promise.reject()

Part 2 - Scheduling

What is Scheduling?

- Managing the triggering of tasks to be run
  - nexting
  - erroring
  - completing
  - subscription

Schedulers can be provided to most Observable creation methods

```
Observable.of(1, 2, 3, scheduler)
Observable.timer(100, scheduler)
Observable.range(0, 10, scheduler)
Observable.from([1, 2, 3], scheduler)
```

…and many more

You can also schedule a pre-existing observable

- subscribeOn(scheduler)
- observeOn(scheduler)

Scheduler API

```
interface Scheduler {
  schedule<T>(
    action: (state?: T) => void,
    delay?: number,
    state?: T
  )
  now(): number
}
```

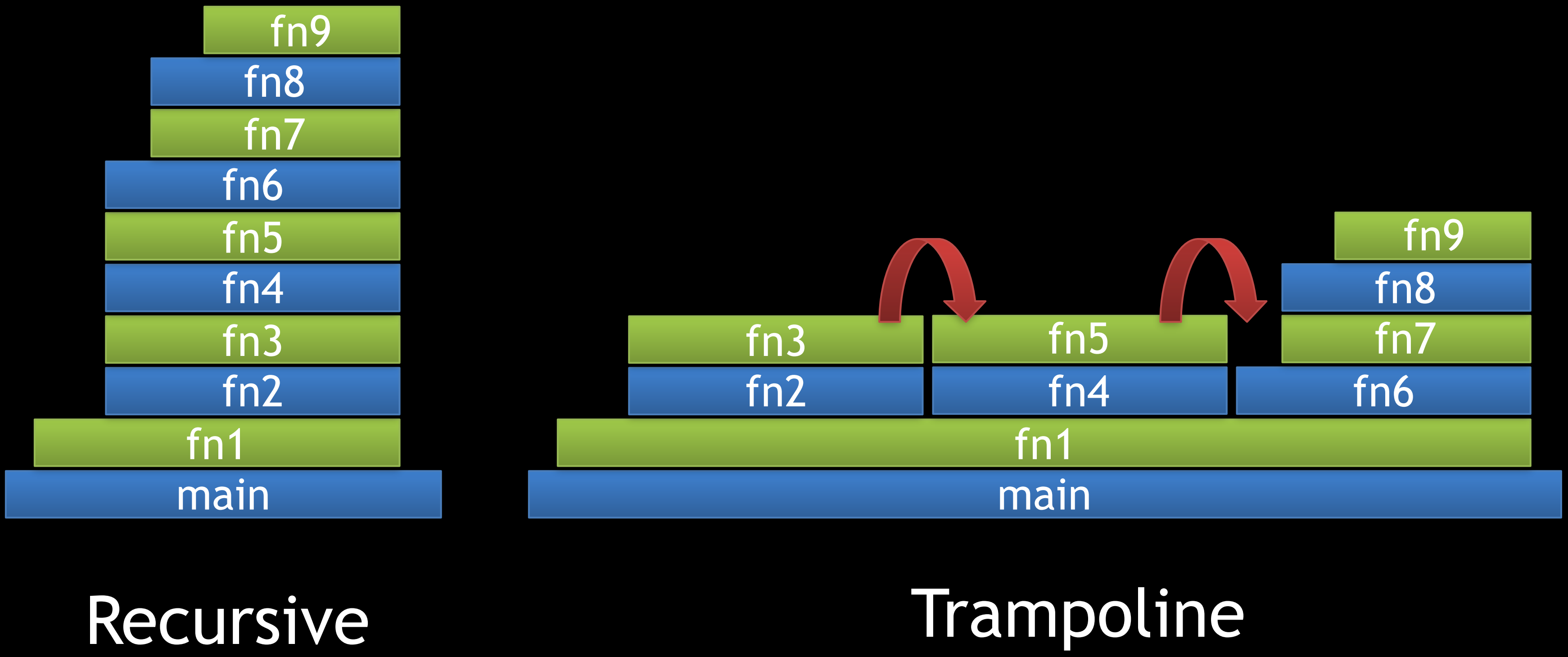
Why?

- Preventing Stack Overflows
- Sometimes to ensure asynchronous behavior
- To coordinate with outside lifecycles
- To enable deterministic tests

Schedulers

- none (default)
- queue
- asap
- async
- animationFrame
- TestScheduler

Scheduling


fn9
fn8
fn7
fn6
fn5
fn4
fn3
fn2
fn1
main
Recursive
fn9
fn8
fn3
fn5
fn7
fn2
fn4
fn6
fn1
main
Trampoline

no scheduler (default)

Delay 0 – immediate execution

Delay > 0 – setTimeout scheduling

queue scheduler (aka trampoline)

- Delay 0 – adds task to queue, if the queue isn’t already being processed, starts processing the queue.
- Delay > 0 – setTimeout scheduling

queue scheduler (basic impl)

```
const queue = [];
let flushing  = false;

function queueSchedule(fn, delay) {
    queue.push(fn);
    if (!flushing) flush();
}

function flush() {
  flushing = true;
  while (queue.length > 0) {
    const fn = queue.shift();
    fn();
  }
  flushing = false;
}
```

asap scheduler

- delay 0
  - aka “next job”, “microtask” or “next tick”
  - Same scheduling as promises
  - before setTimeout(fn, 0)
- delay > 0 uses setTimeout scheduling

async scheduler

- uses setTimeout or setInterval for all scheduling

animationFrame scheduler

- uses requestAnimationFrame to schedule
- An example of using scheduling to coordinate with a lifecycle

TestScheduler

- completely synchronous
- does not execute until `flush` is called
- deterministic tests
- helper functions for creating and asserting test observables
- Used to run > 2000 tests in under 2 seconds for RxJS 5

Part 3 - Operators

Observables are sets

(Just like Iterables)

Sets have operators

Methods that tranform sets

Array filter, map, reduce

```
const source = [1, 2, 3, 4, 5];

const result = source
      .filter(x => x % 2 === 0)
      .map(x => x + ‘!’)
      .reduce((state, x) => state + ‘>’ + x, ‘’);

console.log(result); // “>2!>4!”
```

Try it

exercises/node/array-methods.js

Observables have operators

Methods on observable that return new observables

Most basic example… map

```
Observable.of(‘Ben’)
  .map(name => `Hello, ${name}`)
  .subscribe(x => console.log(x))
```

Try it

exercises/node/operators-basic.js

The anatomy of an operator (basically)

```
Observable.prototype.map = function (project) {
  return new Observable(
    observer => this.subscribe({
      next(value) { observer.next(project(value)); },
      error(err) { observer.error(err); },
      complete() { observer.complete() }
    })
  )
};
```

Operators

- return a new observable
- new observable creates an observer that does the “work” of the operator
- observer is linked to a destination observer “down stream”

What happens when we map to something async though?

```
keyUps.map(e => ajax(url))
    .subscribe(x => console.log(x));
```

```
[object Object]
[object Object]
[object Object]
```

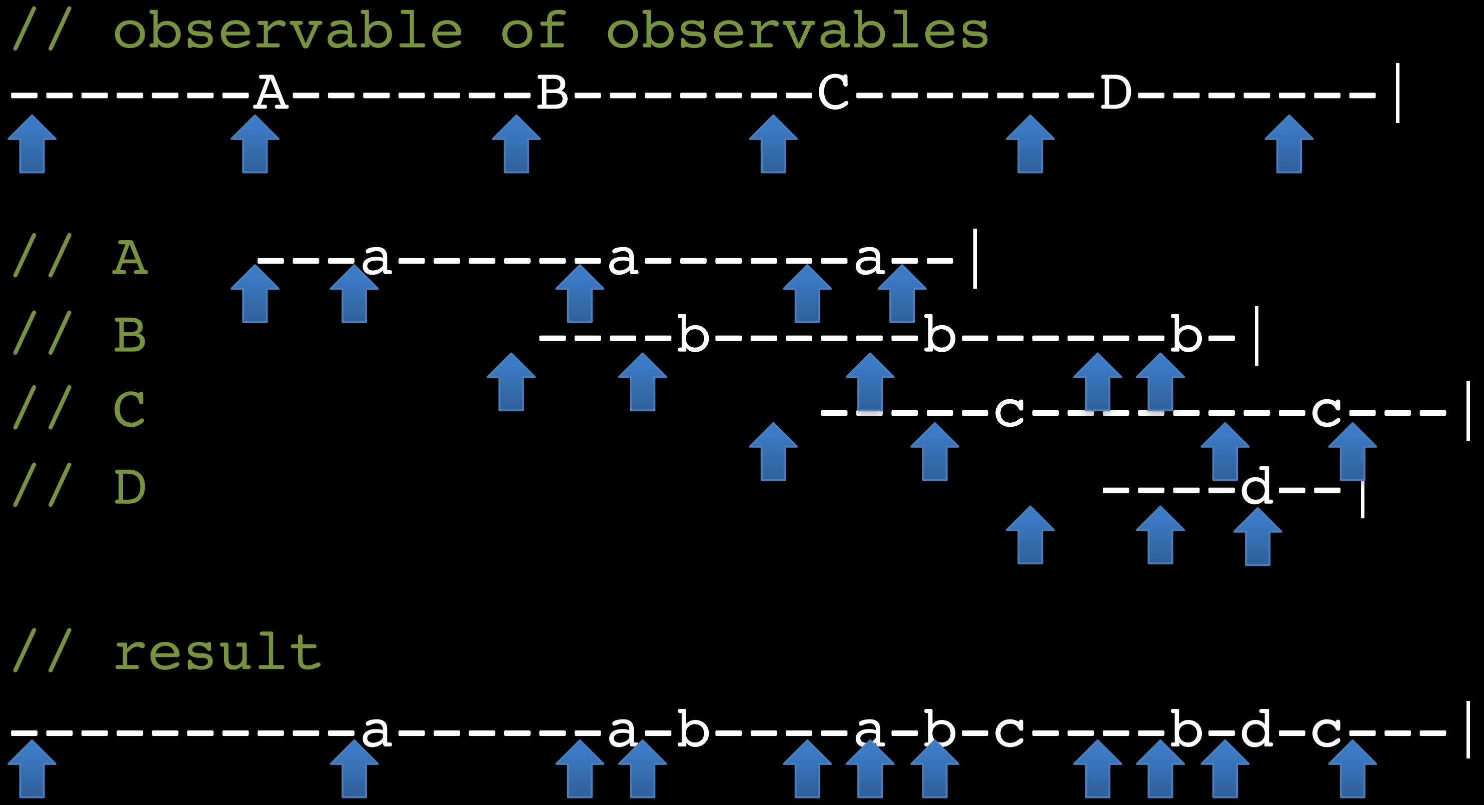
Part 4 - Flattening and Merging

The Three Most Common Strategies

- Merge
- Concat
- Switch

mergeAll

```
const result = observables.mergeAll();
```



```
// observable of observables
-------A-------B-------C-------D-------|

// A   ---a------a------a--|
// B           ----b------b------b-|
// C                   -----c--------c---|
// D                           ----d--|

// result
----------a------a-b----a-b-c----b-d-c---|
```

Merge

- will subscribe to ALL observables
- and forward ALL of their values
- until ALL observables are complete (including the source observable)

Merge Operators

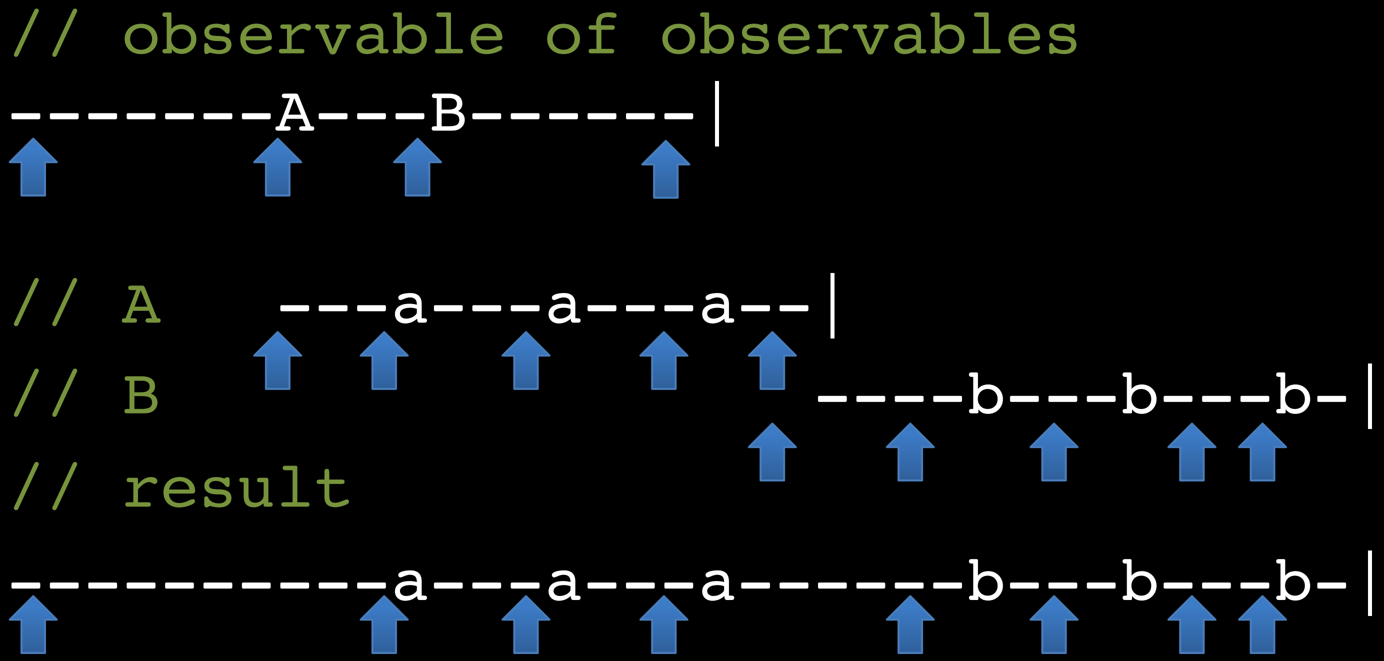
- mergeAll()
- mergeMap(fn) = map(fn).mergeAll()
- a.merge(b, c);
- Observable.merge(a, b, c);

Try it

exercises/node/merge.js

concatAll

```
const result = observables.concatAll();
```



```
// observable of observables
--------A---B-------|

// A   ---a---a---a--|
// B                 ----b---b---b-|
// result
----------a---a---a------b---b---b-|
```

Concat Strategy

- Subscribes to all Observables, but only ONE at a time.
- Other arriving observables wait in a queue and are subscribed as soon as the active one is done.
- Does not complete until all observables complete (including the source)

Concat Operators

- concatAll()
- concatMap(fn) = .map(fn).concatAll()
- a.concat(b, c)
- Observable.concat(a, b, c)

Try it

exercises/node/concat.js

switch

```
const result = observables.switch();

// observable of observables
-------A--------B--------C-----------|

// A   ---a---a                -|
// B            ----b--
// C                     ---c--c----c---|
// result
----------a---a-----b---b---c--c----c---|
```

Switch Strategy

- Subscribes to each observable as soon as it arrives, but only ONE subscription at a time.
- If one arrives while another is active, the active subscription is unsubscribed and thrown away.

Switch Operators

- switch()
- switchMap(fn) = .map(fn).switch()

Try it

exercises/node/switch.js

Part 4 - Observable Chains

Observable Chains

```
let source = Observable.of(1, 2,
3)
   .filter(x => x % 2 === 0)
   .map(x => x + ‘!!!’);
```

Observable Chains

- Return a new observable at each step
- Observables are lazy
- At subscription time, Observables tie an observer to a provider
- In an operator chain, the “provider” is the previous observable (subscription)

Observable Chains

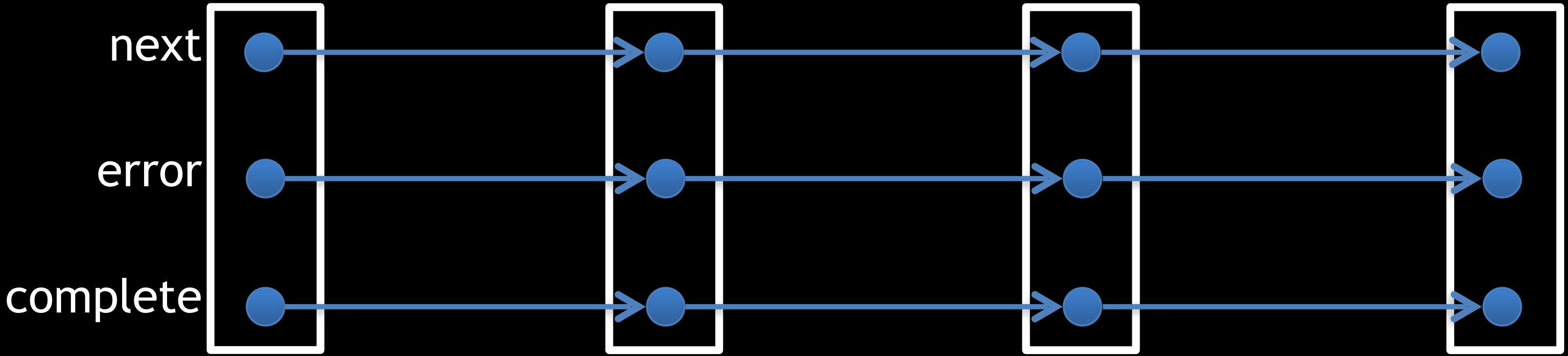
- On subscribe
  - Sets up many observers and chains them together
  - All the way up to the original source provider
  - Ties it all to a single subscription

Observables are really just templates

To set up chains of observers

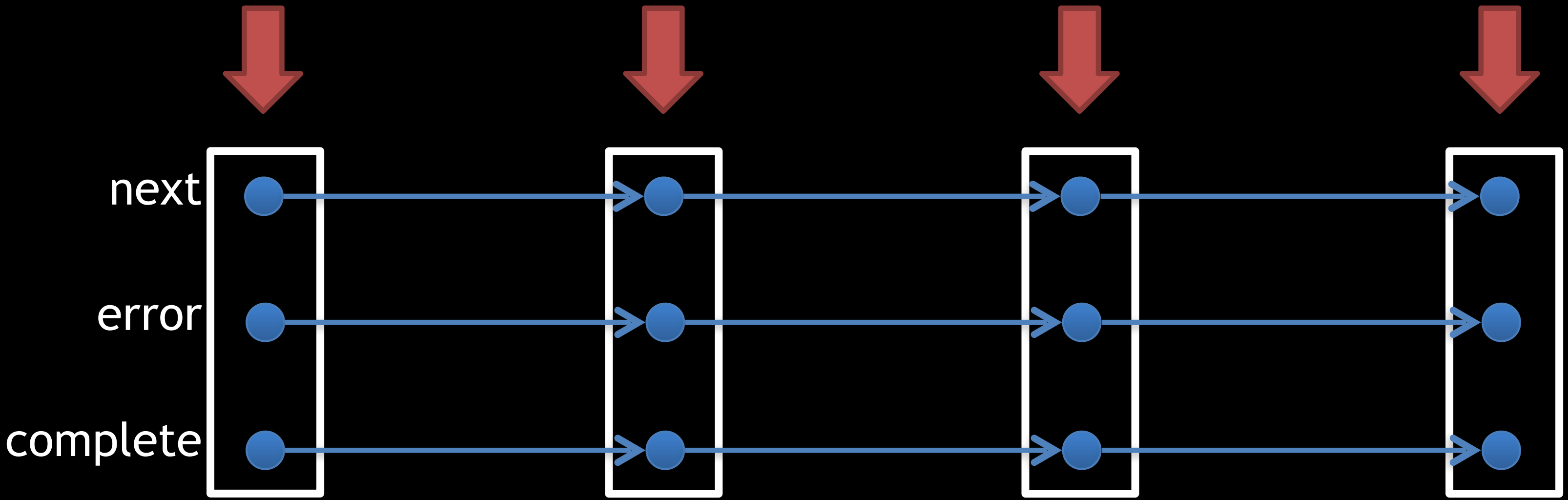
Chain of Observers

```
Observable.interval(1000)
       .filter(x => x % 2 === 0)
       .map(x => x + x)
       .subscribe(nextFn, errorFn, completeFn)
```



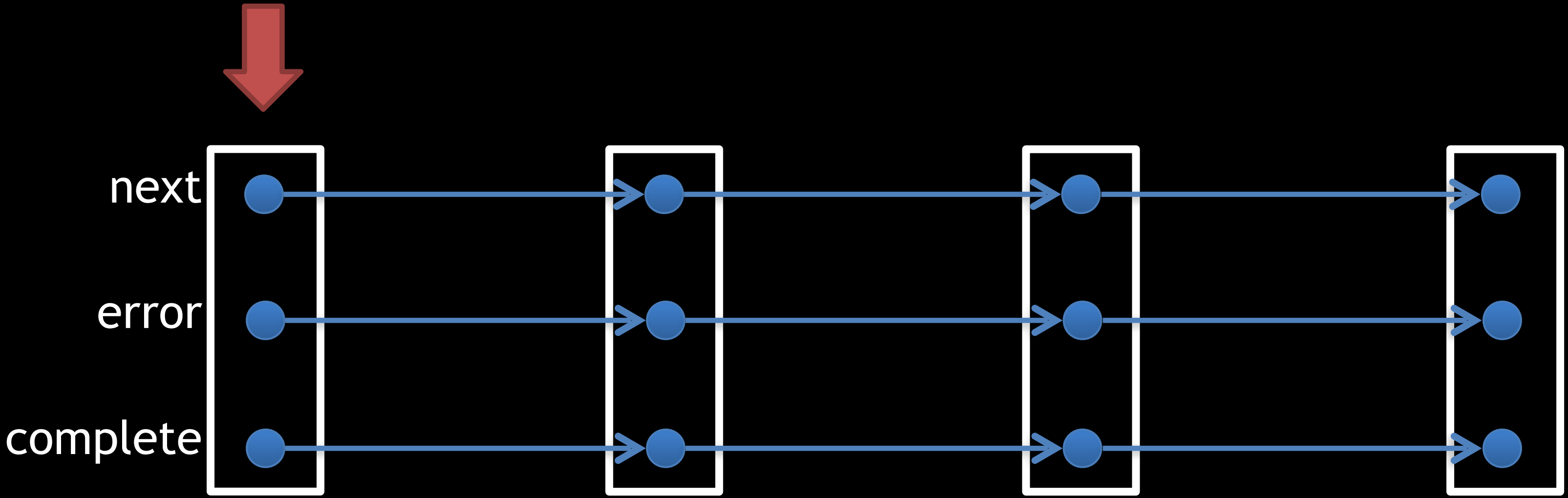
These are observers

```
Observable.interval(1000)
        .filter(x => x % 2 === 0)
        .map(x => x + x)
        .subscribe(nextFn, errorFn, completeFn)
```



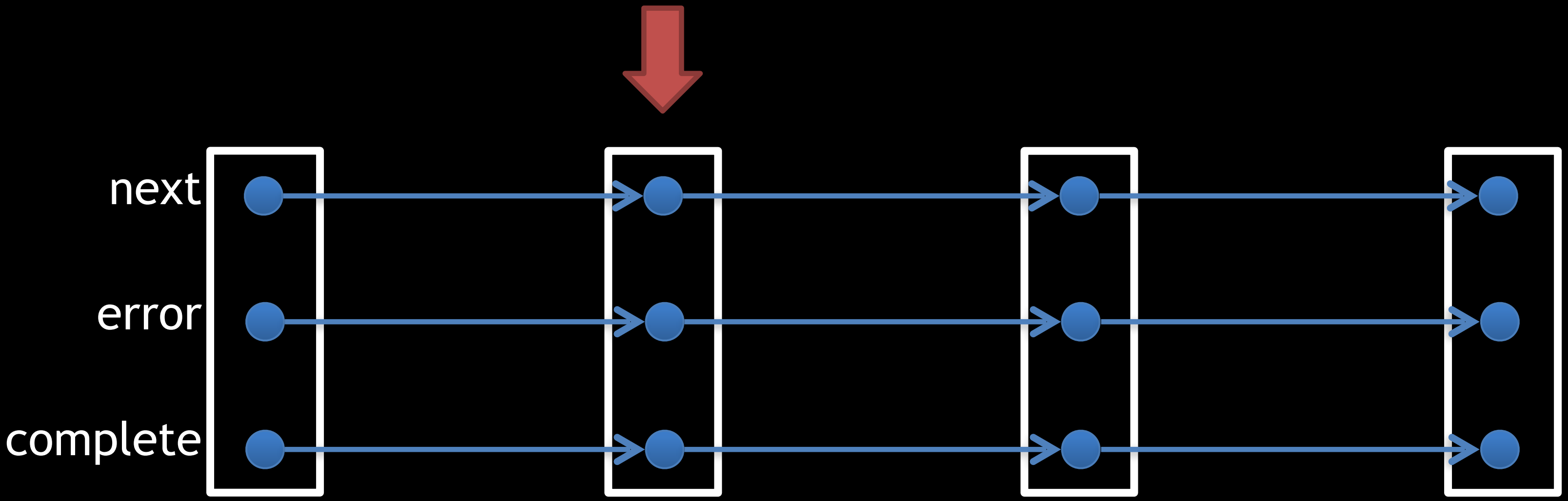
One for the producer at the head

```
Observable.interval(1000)
        .filter(x => x % 2 === 0)
        .map(x => x + x)
        .subscribe(nextFn, errorFn, completeFn)
```



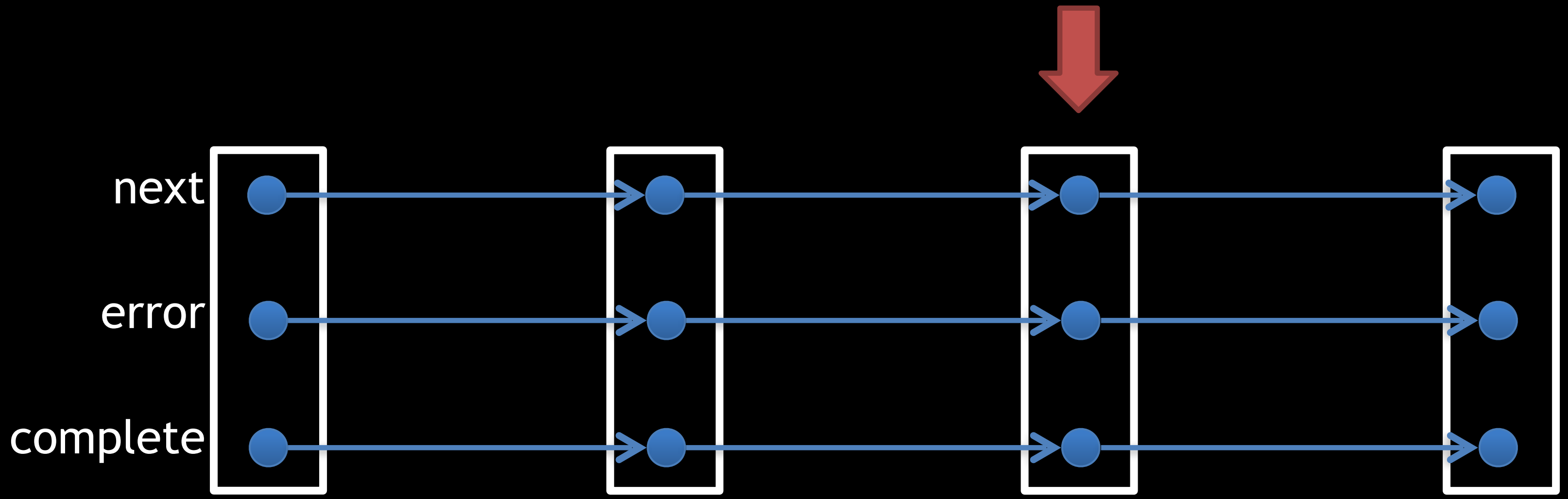
One for your filter operator

```
Observable.interval(1000)
        .filter(x => x % 2 === 0)
        .map(x => x + x)
        .subscribe(nextFn, errorFn, completeFn)
```



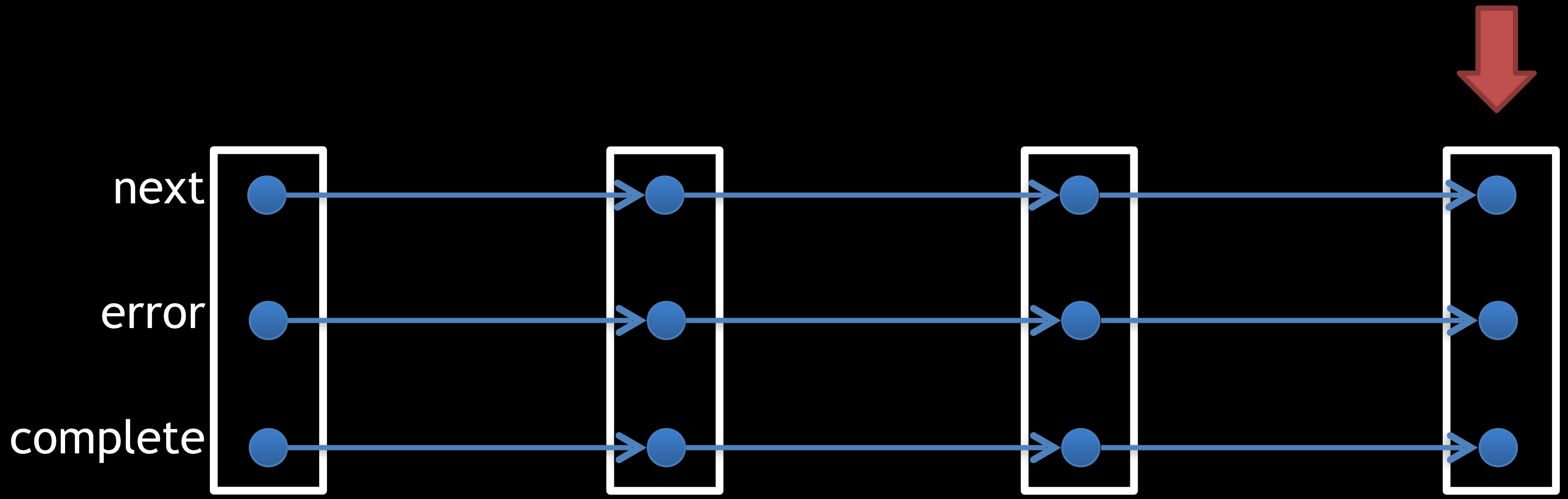
One for your map operator

```
Observable.interval(1000)
        .filter(x => x % 2 === 0)
        .map(x => x + x)
        .subscribe(nextFn, errorFn, completeFn)
```



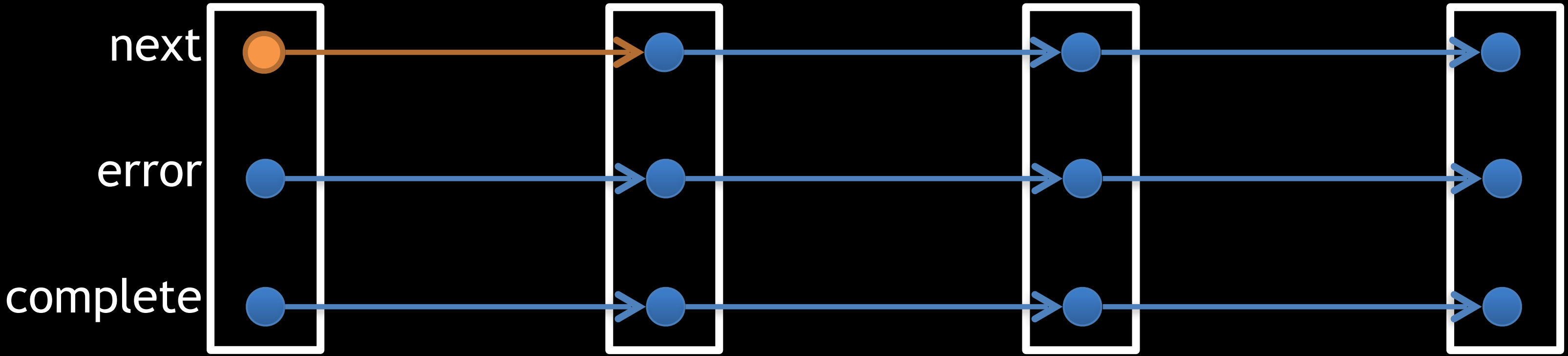
One at the tail that wraps your handlers

```
Observable.interval(1000)
          .filter(x => x % 2 === 0)
          .map(x => x + x)
          .subscribe(nextFn, errorFn, completeFn)
```



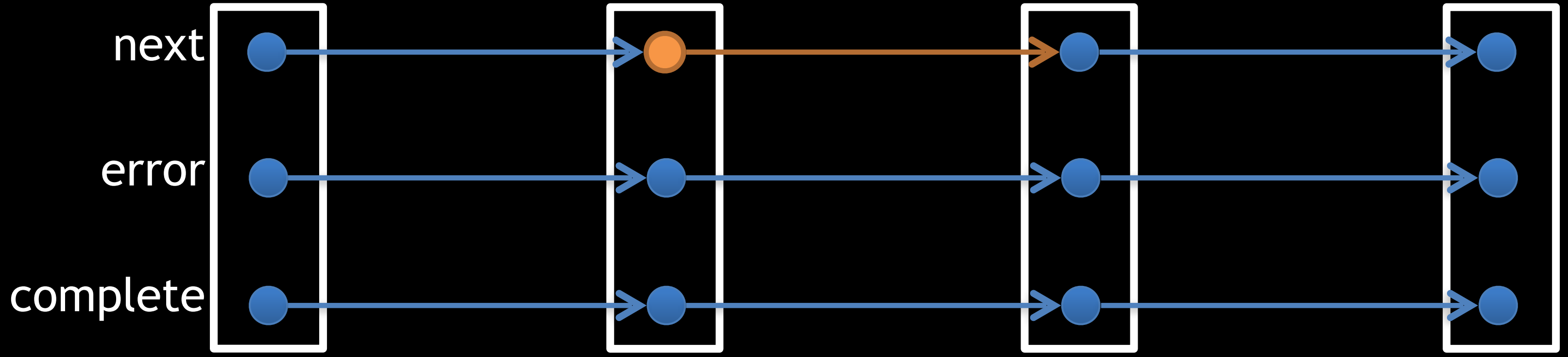
The producer nexts `0` to the filter

```
Observable.interval(1000)
          .filter(x => x % 2 === 0)
          .map(x => x + x)
          .subscribe(nextFn, errorFn, completeFn)
```



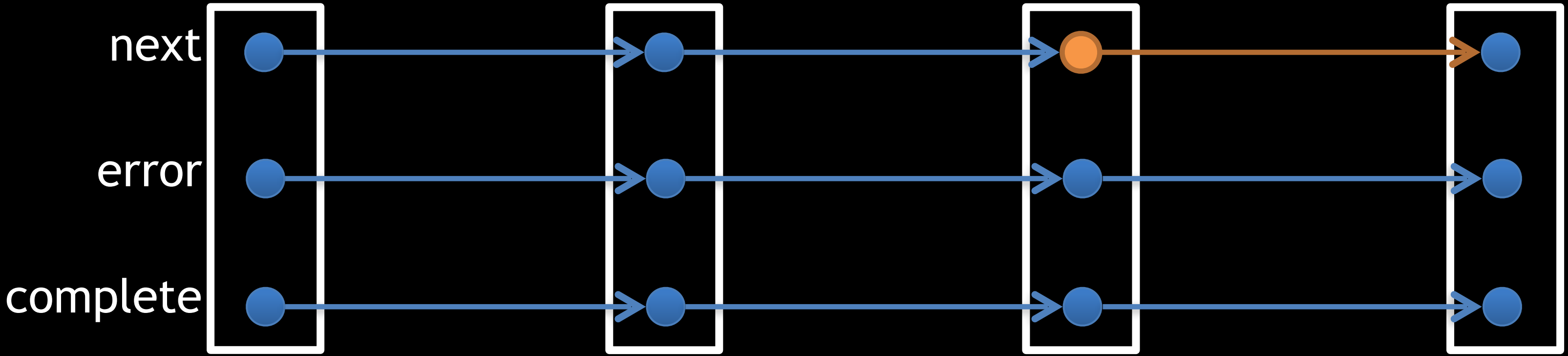
The filter passes for `0` and nexts to map

```
Observable.interval(1000)
          .filter(x => x % 2 === 0)
          .map(x => x + x)
          .subscribe(nextFn, errorFn, completeFn)
```



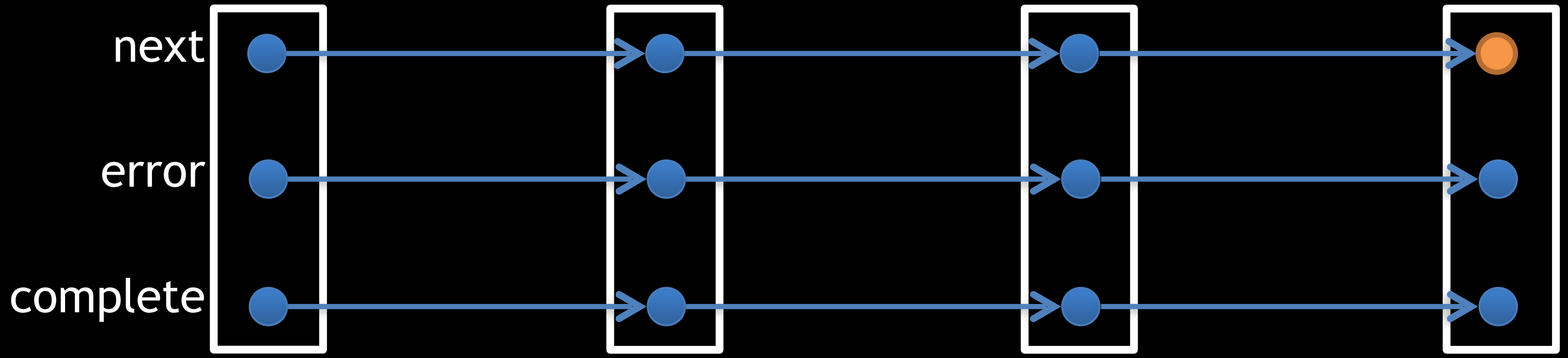
The value is mapped and sent along

```
Observable.interval(1000)
         .filter(x => x % 2 === 0)
         .map(x => x + x)
         .subscribe(nextFn, errorFn, completeFn)
```



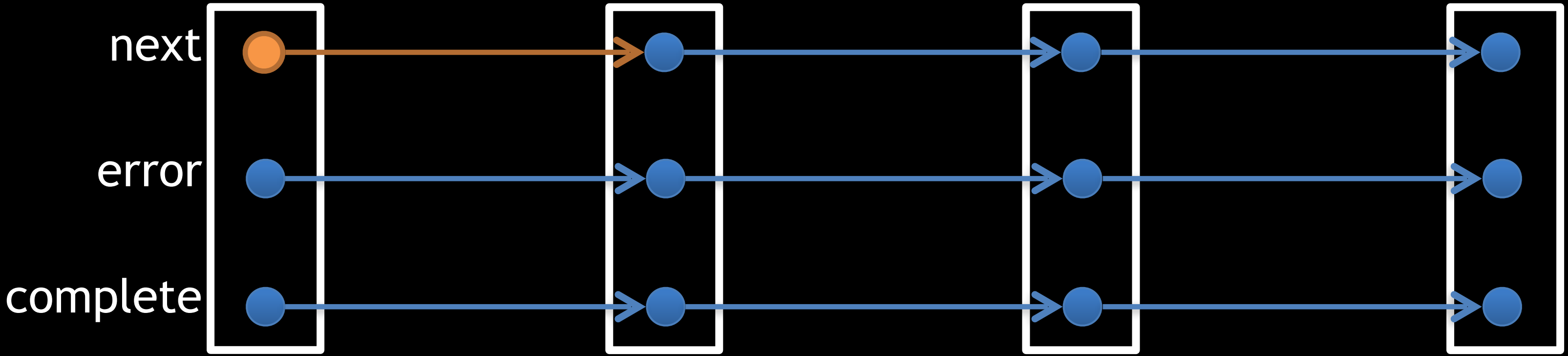
The value then hits the next handler

```
Observable.interval(1000)
          .filter(x => x % 2 === 0)
          .map(x => x + x)
          .subscribe(nextFn, errorFn, completeFn)
```



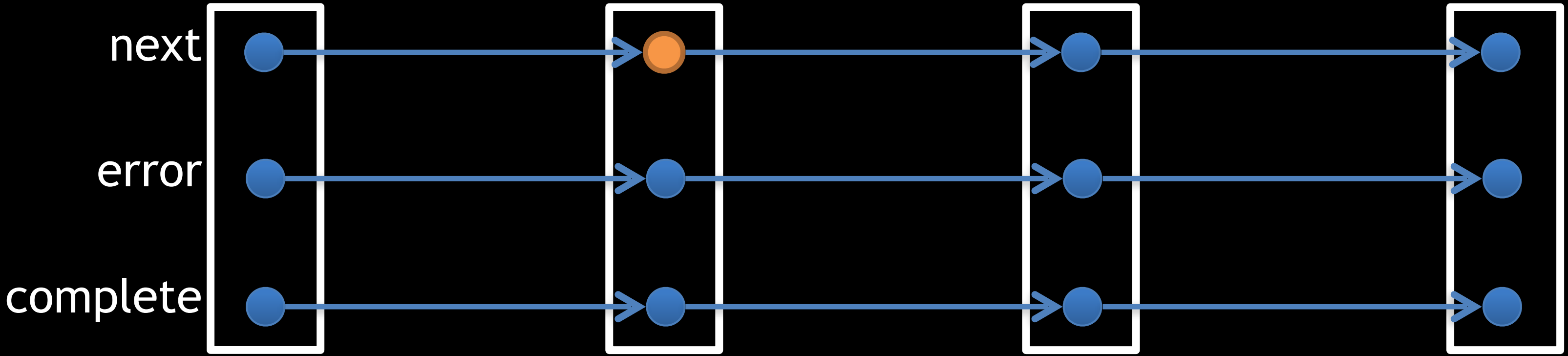
Next we’re sending along a 1

```
Observable.interval(1000)
        .filter(x => x % 2 === 0)
        .map(x => x + x)
        .subscribe(nextFn, errorFn, completeFn)
```



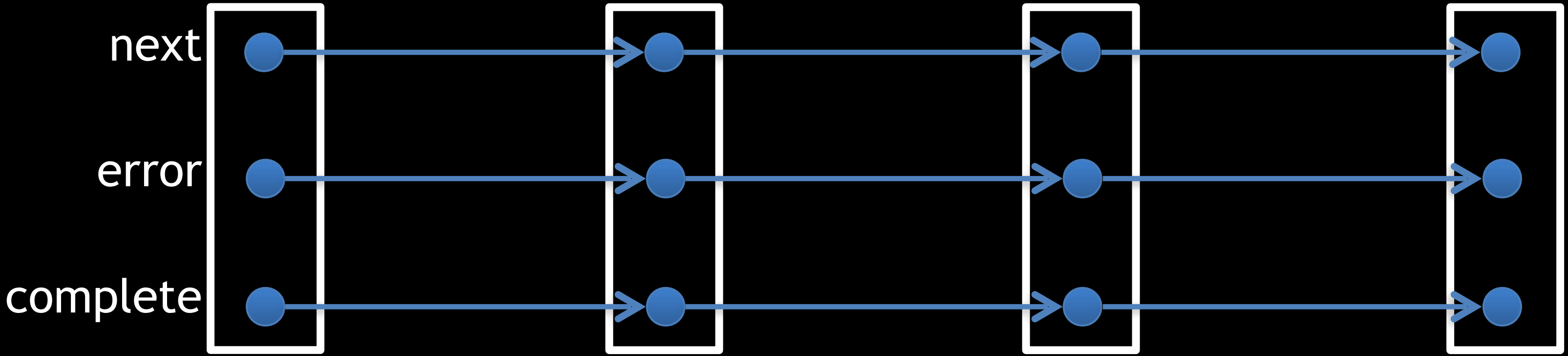
But the value 1 doesn't pass the filter

```
Observable.interval(1000)
         .filter(x => x % 2 === 0)
         .map(x => x + x)
         .subscribe(nextFn, errorFn, completeFn)
```



… so it’s not sent along to map.

```
Observable.interval(1000)
        .filter(x => x % 2 === 0)
        .map(x => x + x)
        .subscribe(nextFn, errorFn, completeFn)
```



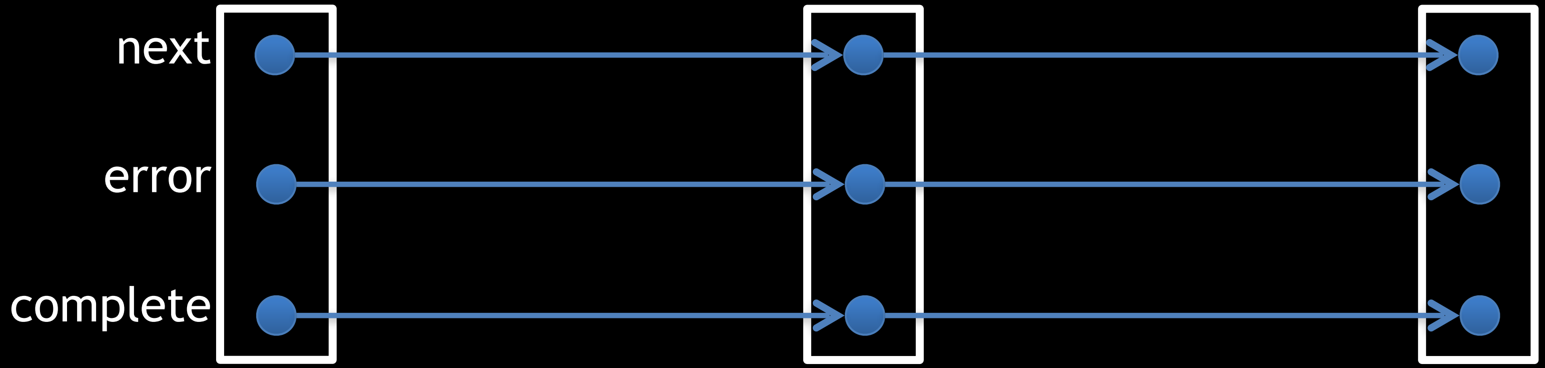
Part 5 - Error Handling

Observers will no longer pass along values after:

- they receive an error
- they receive a completion
- the accompanying subscription is unsubscribed

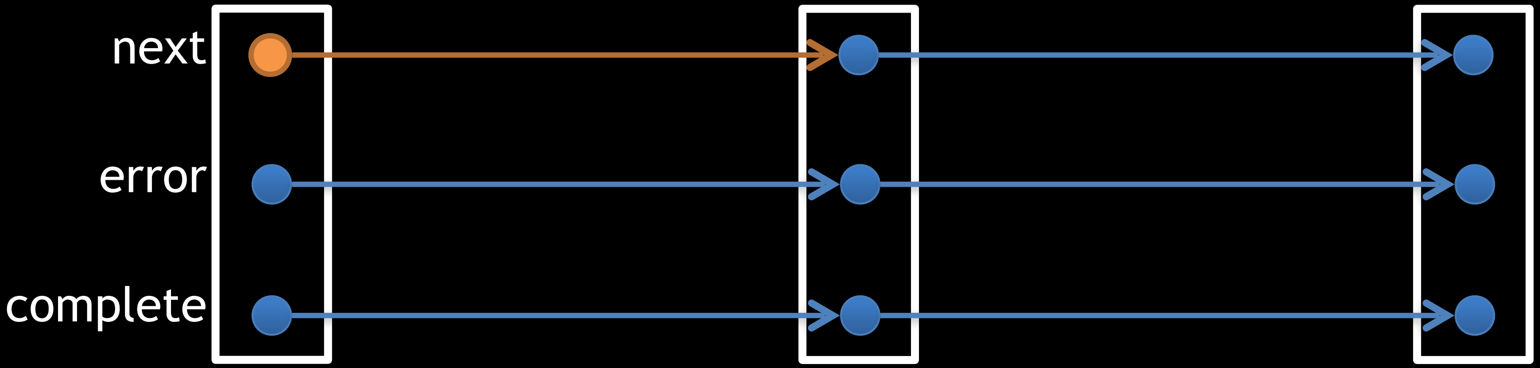
What does that mean for error handling?

```
Observable.interval(1000)
  .map(x => {
    if (x === 1) throw new Error ('haha');
    return x;
  })
  .subscribe(nextFn, errorFn, completeFn);
```



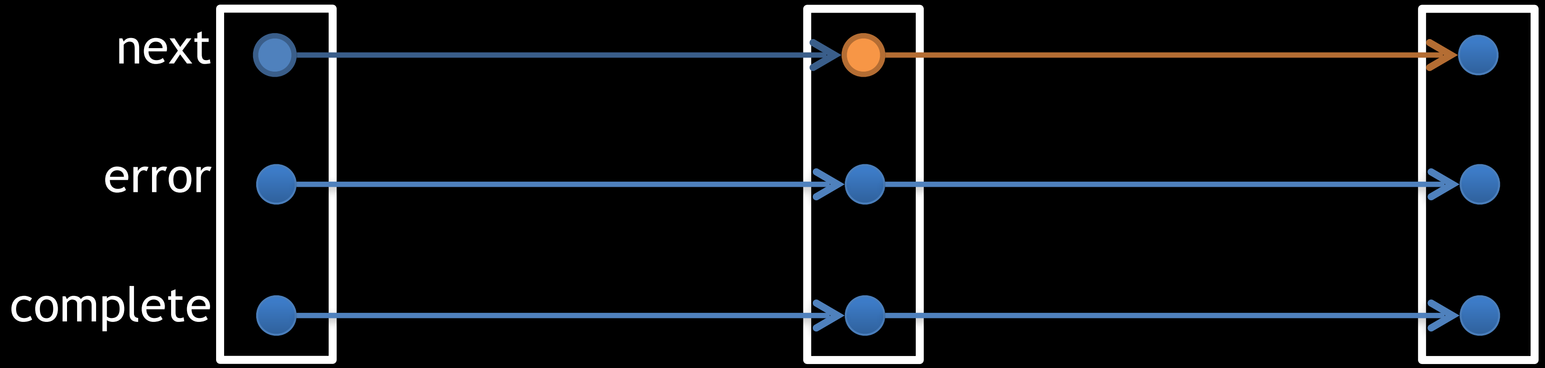
sending 0

```
Observable.interval(1000)
  .map(x => {
    if (x === 1) throw new Error ('haha');
    return x;
  })
  .subscribe(nextFn, errorFn, completeFn);
```



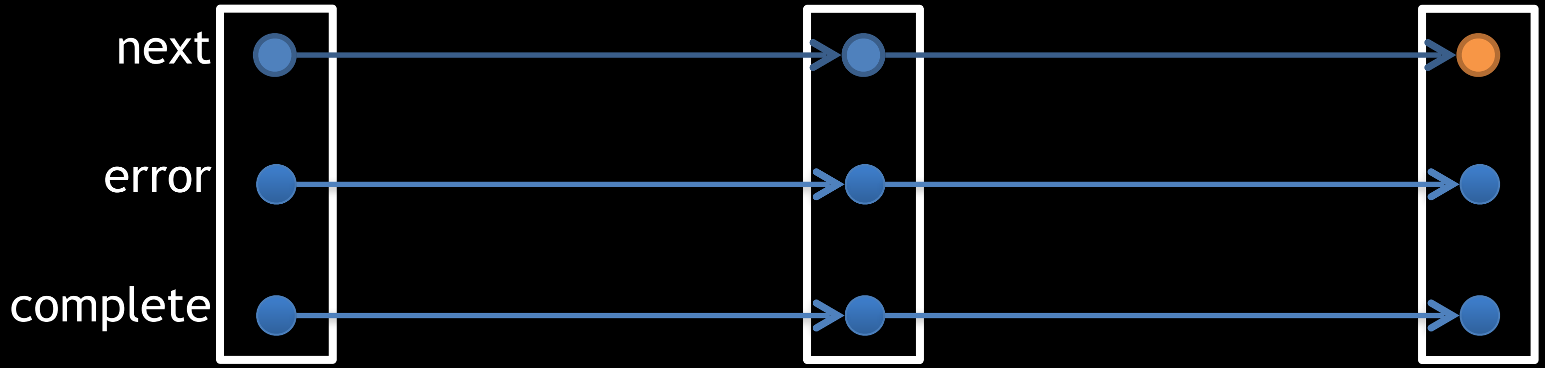
sending 0

```
Observable.interval(1000)
   .map(x => {
     if (x === 1) throw new Error ('haha');
     return x;
   })
   .subscribe(nextFn, errorFn, completeFn);
```



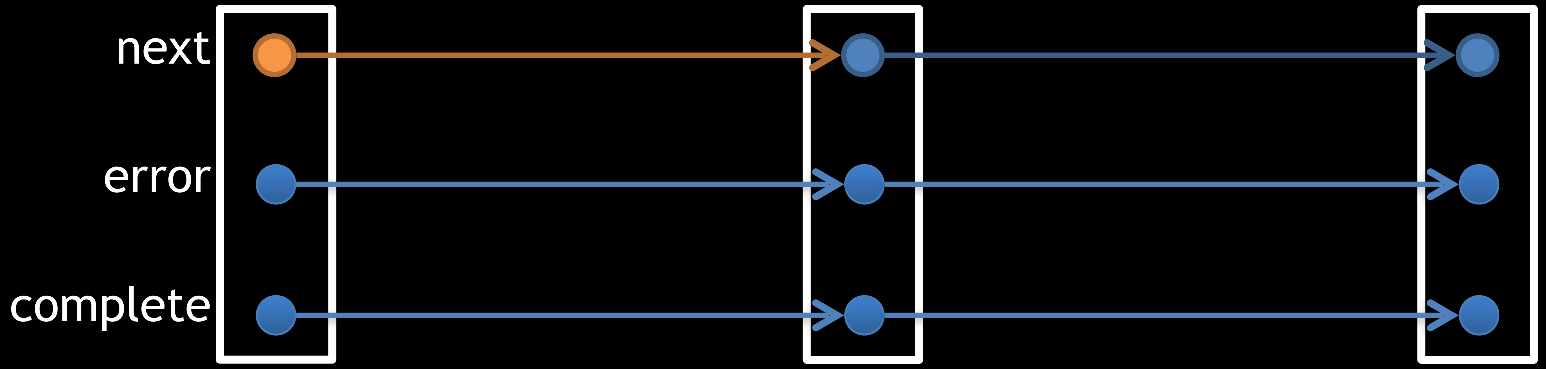
sending 0

```
Observable.interval(1000)
  .map(x => {
    if (x === 1) throw new Error ('haha');
    return x;
  })
  .subscribe(nextFn, errorFn, completeFn);
```



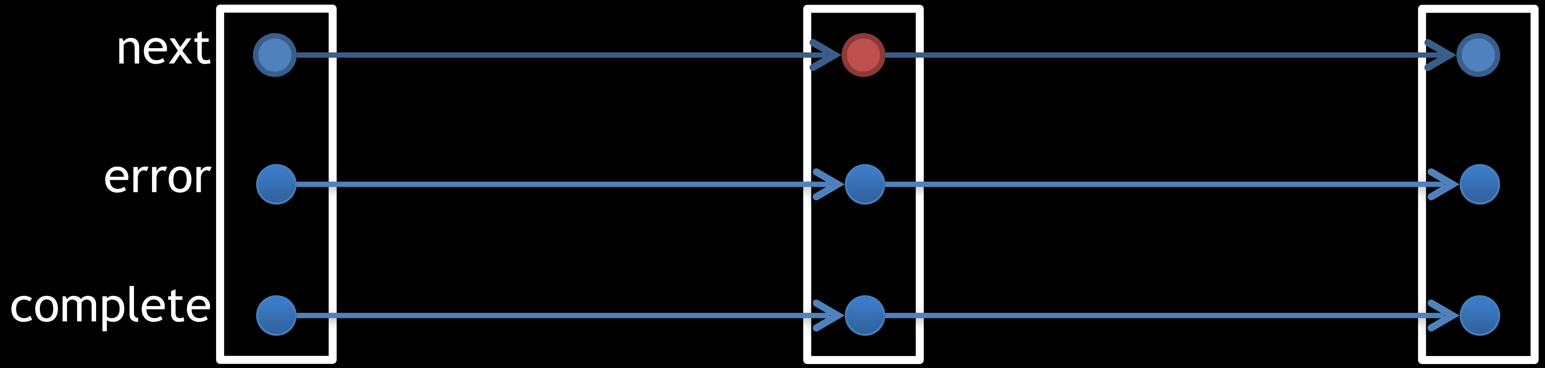
sending 1

```
Observable.interval(1000)
  .map(x => {
    if (x === 1) throw new Error ('haha');
    return x;
  })
  .subscribe(nextFn, errorFn, completeFn);
```



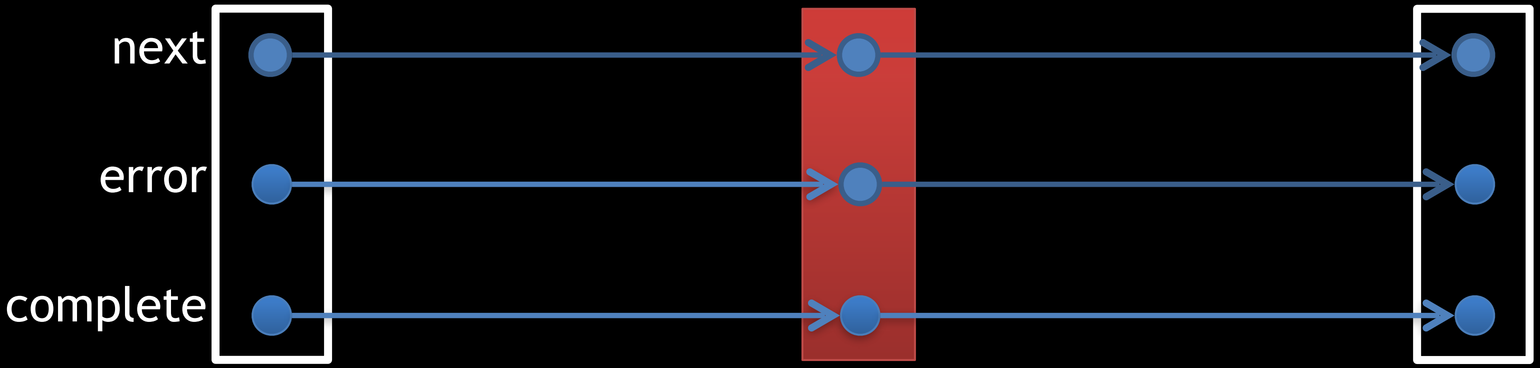
1 throws in our map!

```
Observable.interval(1000)
  .map(x => {
    if (x === 1) throw new Error ('haha');
    return x;
  })
  .subscribe(nextFn, errorFn, completeFn);
```



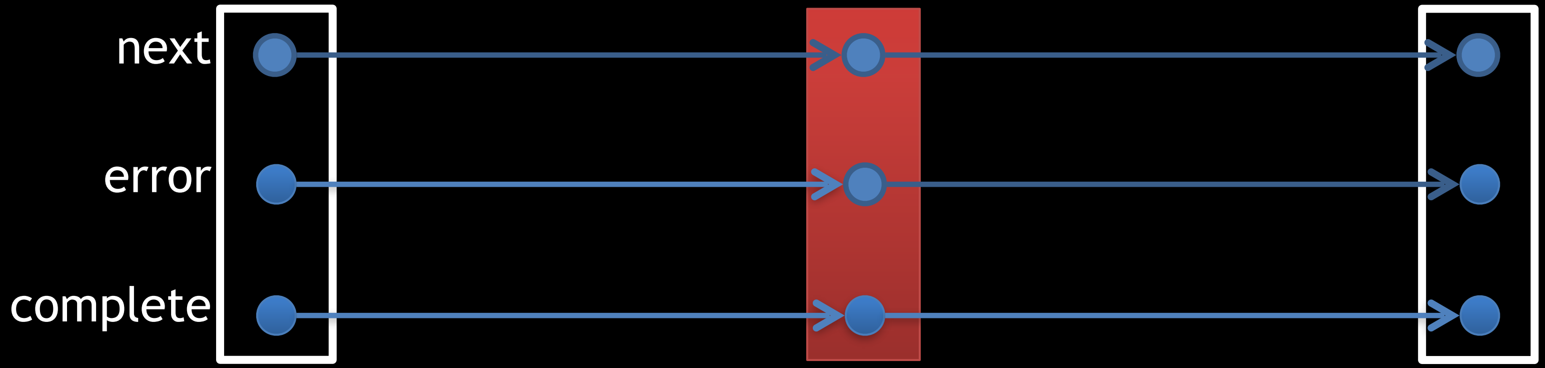
… so the observer is rendered inert.

```
Observable.interval(1000)
   .map(x => {
     if (x === 1) throw new Error ('haha');
     return x;
   })
   .subscribe(nextFn, errorFn, completeFn);
```



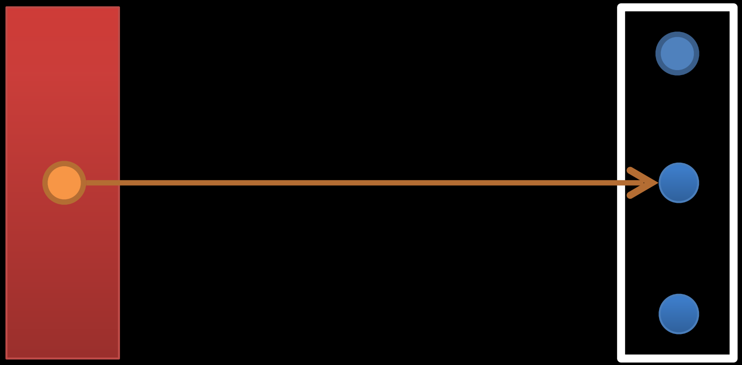
(that means nothing else can pass through it)

```
Observable.interval(1000)
   .map(x => {
     if (x === 1) throw new Error ('haha');
     return x;
   })
   .subscribe(nextFn, errorFn, completeFn);
```



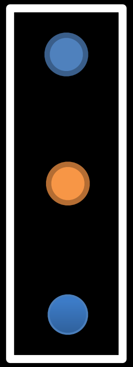
… and an error is signaled down the chain

```
Observable.interval(1000)
   .map(x => {
     if (x === 1) throw new Error ('haha');
     return x;
   })
   .subscribe(nextFn, errorFn, completeFn);
```



… and an error is signaled down the chain

```
Observable.interval(1000)
  .map(x => {
    if (x === 1) throw new Error ('haha');
    return x;
  })
  .subscribe(nextFn, errorFn, completeFn);
```



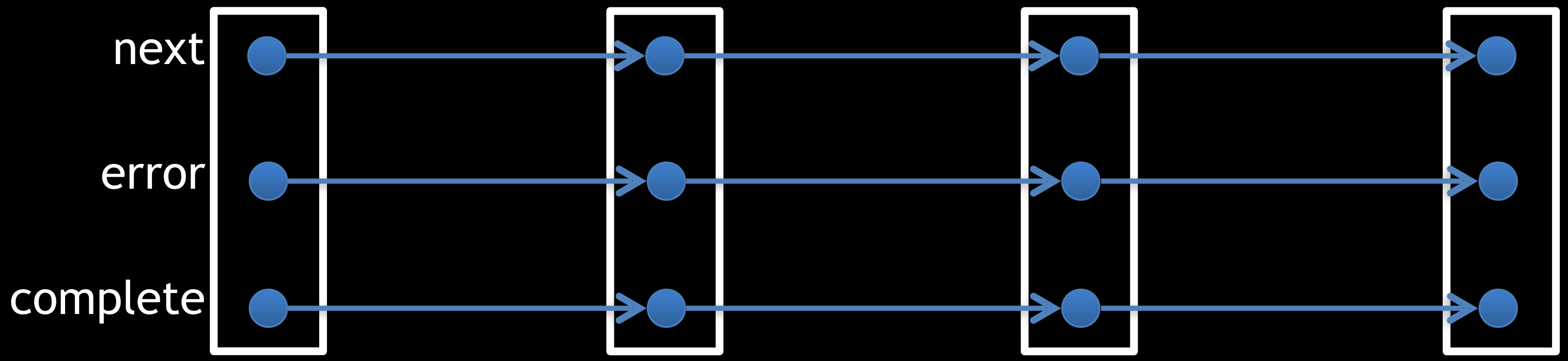
The `catch` operator

- Takes a function that gives you and error and expects you to return an observable
- Very similar to promise `catch`

```
Observable.of(1)
      .map(someFn)
      .catch(err => Observable.of(‘this is fine’))
      .subscribe(nextFn, errorFn, completeFn);
```

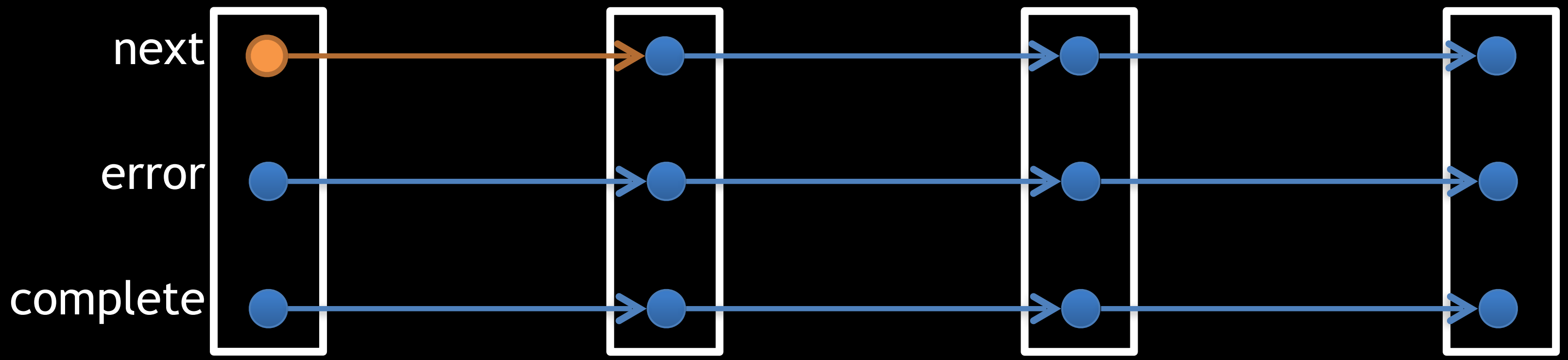
Using catch

```
Observable.of(1)
    .map(x => {
        throw new Error(‘I hate ones!’);
    })
    .catch(err => Observable.of(2))
    .subscribe(nextFn, errorFn, completeFn)
```



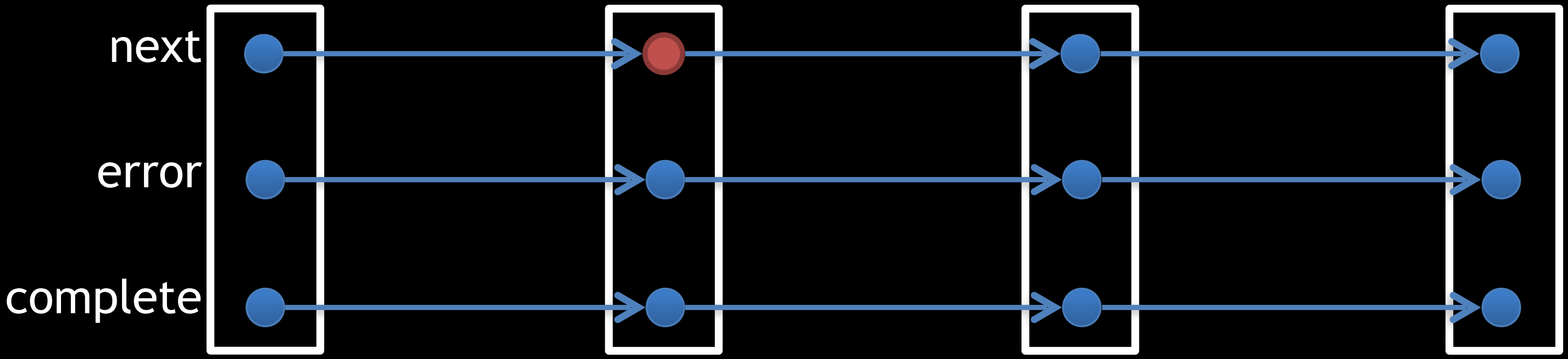
Send the 1

```
Observable.of(1)
    .map(x => {
        throw new Error(‘I hate ones!’);
    })
    .catch(err => Observable.of(2))
    .subscribe(nextFn, errorFn, completeFn)
```



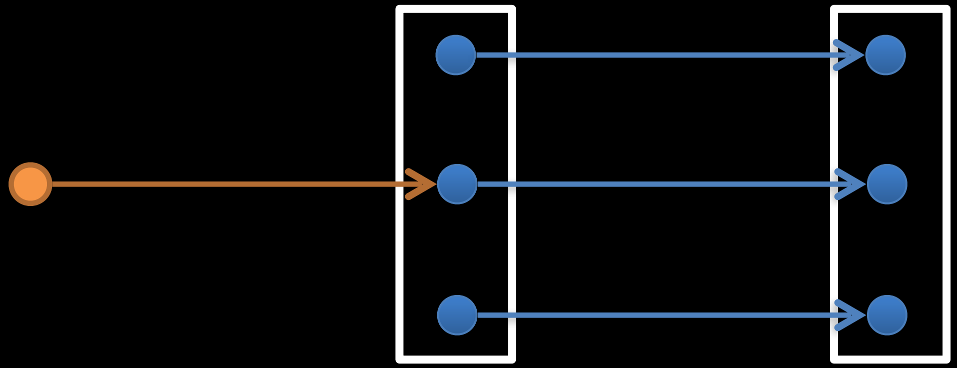
Uh oh... error!

```
Observable.of(1)
    .map(x => {
        throw new Error(‘I hate ones!’);
    })
    .catch(err => Observable.of(2))
    .subscribe(nextFn, errorFn, completeFn)
```



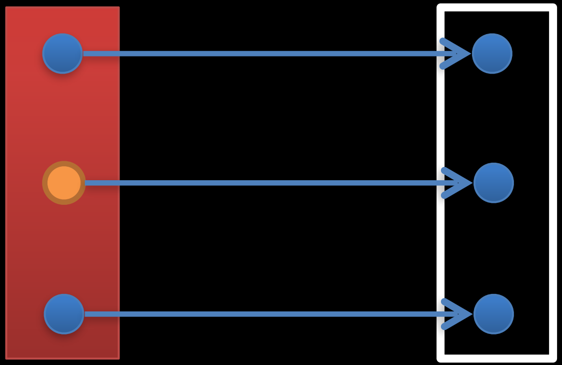
Send the error to `catch`, observers from this point and up are “dead”

```
Observable.of(1)
     .map(x => {
         throw new Error(‘I hate ones!’);
     })
     .catch(err => Observable.of(2))
     .subscribe(nextFn, errorFn, completeFn)
```



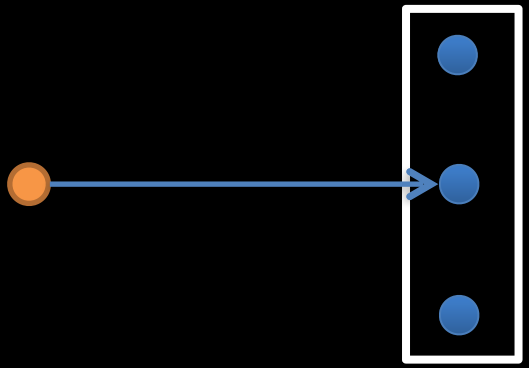
The `catch` observer got an error, so it’s actually “dead” too.

```
Observable.of(1)
     .map(x => {
          throw new Error(‘I hate ones!’);
     })
     .catch(err => Observable.of(2))
     .subscribe(nextFn, errorFn, completeFn)
```



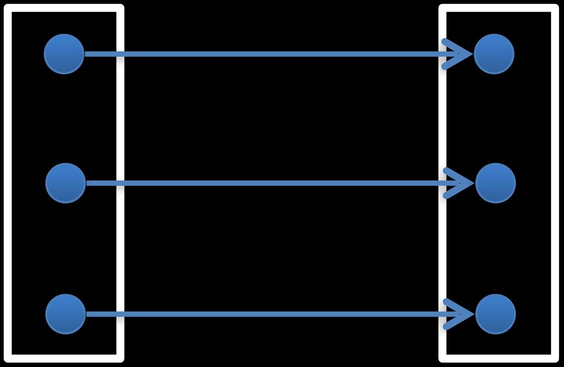
But the error path in `catch` will map to a new Observable!

```
Observable.of(1)
    .map(x => {
        throw new Error('I hate ones!');
    })
    .catch(err => Observable.of(2))
    .subscribe(nextFn, errorFn, completeFn)
```



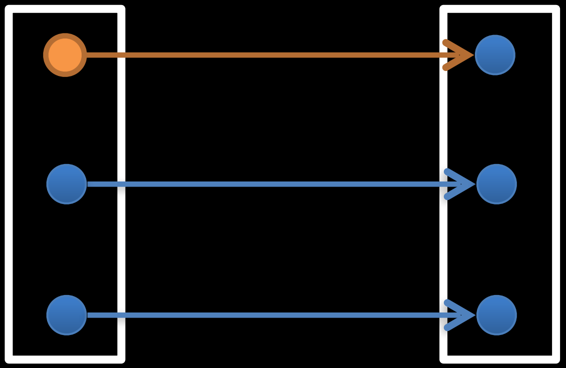
Which is subscribed to with an observer

```
Observable.of(1)
    .map(x => {
        throw new Error(‘I hate ones!’);
    })
    .catch(err => Observable.of(2))
    .subscribe(nextFn, errorFn, completeFn)
```



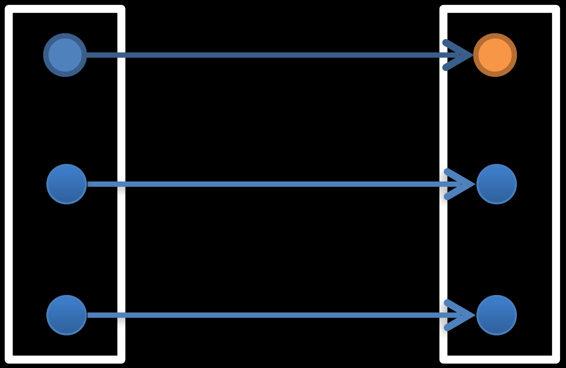
Signaling a 2

```
Observable.of(1)
    .map(x => {
        throw new Error(‘I hate ones!’);
    })
    .catch(err => Observable.of(2))
    .subscribe(nextFn, errorFn, completeFn)
```



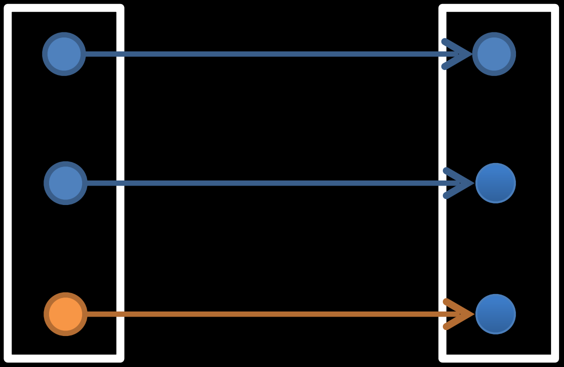
to the next handler

```
Observable.of(1)
    .map(x => {
        throw new Error(‘I hate ones!’);
    })
    .catch(err => Observable.of(2))
    .subscribe(nextFn, errorFn, completeFn)
```



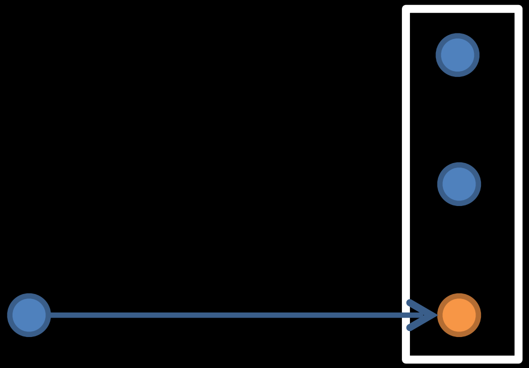
then completing (because `of`)

```
Observable.of(1)
    .map(x => {
        throw new Error('I hate ones!');
    })
    .catch(err => Observable.of(2))
    .subscribe(nextFn, errorFn, completeFn)
```



and we’re all done.

```
Observable.of(1)
    .map(x => {
        throw new Error(‘I hate ones!’);
    })
    .catch(err => Observable.of(2))
    .subscribe(nextFn, errorFn, completeFn)
```



Try it

exercises/node/first-catch.js

But catching still allows the source to die…

What if I want this interval to continue?

```
Observable.interval(1000)
    .map(x => {
        if (x === 4) throw new Error(‘no fours’);
        return x + ‘!’;
    })
    .catch(err => Observable.of(‘nah, 4 is okay’))
    .subscribe(nextFn, errorFn, completeFn)

// “0!”
// “1!”
// “2!”
// “3!”
// “nah, 4 is okay”
// done
```

Part 6 – Isolating Observer Chains

We can use other observables to set up alternate observer chains

Using a merge operator to create an isolated observer chain

```
Observable.interval(1000)
  .mergeMap(x =>
    Observable.of(x)
      .map(x => {
        if (x === 4) throw new Error(‘no fours’);
        return x + ‘!’;
      })
      .catch(err => Observable.of(‘nah, 4 is okay’))
  )
  .subscribe(nextFn, errorFn, completeFn)
```

Isolate what can fail to it’s own observable

```
Observable.interval(1000)
  .mergeMap(x =>
    Observable.of(x)
      .map(x => {
        if (x === 4) throw new Error(‘no fours’);
        return x + ‘!’;
      })
      .catch(err => Observable.of(‘nah, 4 is okay’))
  )
  .subscribe(nextFn, errorFn, completeFn)
```

… protecting what you don’t want to fail

```
Observable.interval(1000)
  .mergeMap(x =>
    Observable.of(x)
      .map(x => {
        if (x === 4) throw new Error(‘no fours’);
        return x + ‘!’;
      })
      .catch(err => Observable.of(‘nah, 4 is okay’))
  )
  .subscribe(nextFn, errorFn, completeFn)
```

Try it

exercises/node/isolation.js

A little vocabulary

“side effect”

Code that updates state outside of it’s scope

```
var state = 0;

var fn = () => state++;

// or even

var fn2 = () => console.log(‘wee’);
```

In Rx, side effects can be added anywhere

```
Observable.of(1).map(x => {
  console.log(‘I am a side effect!’);
  return x + 1;
})
.subscribe(x => console.log(‘side effect’, x))
```

But generally, we use `do` or `subscribe` for side effects

```
Observable.of(1)
.do(() => console.log(‘I am a side effect!’))
.map(x => x + 1)
.subscribe(x => console.log(‘side effect’, x))
```

Observable subscriber functions are also a valid place for side effects

```
new Observable((observer) => {
  console.log(‘side effect!’);
  observer.next(‘hi’);
  observer.complete();
});
```

Common side effects

- Updating a variable in an outer scope
- Logging
- Persisting data
- AJAX
- DOM updates
- … and many more

Pure functions

- No side effects
- Does not mutate input
- input determines output 100% of the time

Why is “purity” good?

Caching results for performance gains later… and many other reasons.

Recommended Reading:

https://drboolean.gitbooks.io/mostly-adequate-guide/content/ch3.html

Cold vs Hot

… warm?

Observables are “cold” and “lazy”

```
let starts = 0;

const cold = new Observable(observer => {
  starts++; // side effect to count starts
  const id = setTimeout(() => {
    observer.next('hi');
    observer.complete();
  }, 500);
  return () => clearTimeout();
});

cold.subscribe(x => console.log('next', x));
console.log('starts:', starts);
cold.subscribe(x => console.log('next', x));
console.log('starts:', starts);
```

Making a “hot” observable

```
let starts = 0;

const cold = new Observable(observer => {
  starts++; // side effect to count starts
  const id = setTimeout(() => {
    observer.next('hi');
    observer.complete();
  }, 500);
  return () => clearTimeout();
});


const hot = cold.share();

hot.subscribe(x => console.log('next', x));
console.log('starts:', starts);
hot.subscribe(x => console.log('next', x));
console.log('starts:', starts);
```

TRY IT

exercises/node/cold-vs-hot.js

What happened with the cold/sync observable???

(It synchronously completed the hot observable before the next subscription to it)

I’ll tell you the workaround,

but first…

Subjects!

Subjects

- Observer Pattern
- Register multiple observers
- Observer on one side
- Observable on the other
- No longer usable once closed

Basics

```
var subject = new Subject();

subject.subscribe(
  x => console.log(x),
  err => console.error(err),
  () => console.info(‘done’)
);

subject.next(1);
subject.next(2);
subject.next(3);
subject.complete();
```

TRY IT

exercises/node/my-first-subject.js

Take aways

- Subjects pass values through as an Observable
- Subjects multicast
- Once a Subject completes or errors, it’s silently unusable. (nexting ceases to work)

Subjects can be used as Observers!

Subjects as an Observer

```
const source = Observable.timer(1000)
    .mapTo(‘hello there’);

const subject = new Subject();

subject.subscribe(x => console.log(x));

source.subscribe(subject);
```

Try it

exercises/node/subject-observer.js

Subjects: Two ways to unsubscribe

- subscription.unsubscribe(): removes an individual observer from a subject, but the subject stays “alive”
- subject.unsubscribe(): removes all observers from subject, “killing” it. Subsequent subscriptions will error.

Try it

exercises/node/subject-unsubscribe.js

Takeaway: Killing Subjects

- Subjects are immutable
- Once they’re done, they’re done
- \`complete\` and \`error\` will kill a subject without causing future interactions to error
- \`subject.unsubscribe()\` will kill a subject and cause future interactions to error
- unsubscribing from subscriptions that consume the subject will not kill the subject

In RxJS 5, operators on Subjects return Subjects

```
var subject = new Subject();

var mapped = subject.map(x => x + x);

mapped.subscribe(x => console.log(x));
mapped.subscribe(x => console.log(x));

subject.next(1);
subject.next(2);
subject.next(3);
subject.complete();
```

What are Subjects used for?

- EVERYTHING!

What are Subjects used for?

- ~~EVERYTHING!~~
- Multicasting
- As an adapter

Multicast

(verb) to send data to multiple users across a computer network at the same time

Subject subscription

Adds an observer to a list of observers to notify

Multicasting

- Using `multicast` operator or some derivative
- `publish()`
- `publishReplay()`
- `share()`

Multicast

```
var subject = new Subject();

// Tie source observable into `subject` and have
// all subscribers to the returned observable register
// on that subject.
var connectable = sourceObservable.multicast(subject);

// subscribe a few times
connectable.subscribe(x => console.log(1, x));
connectable.subscribe(x => console.log(2, x));

// calling `connect()` subscribes `subject` to the
// `sourceObservable` and makes it “live”
connectable.connect();
```

“Cold” Observable

- On subscription
  - Create data producer
  - Connect data producer to observer
- On unsubscription
  - Tear down data producer
- Don’t share data producer with other observables

“Hot” Observable

- Subscription closes over previously created data producer.
- unsubscription does not tear down data producer.

“Cold” Observable

```
const cold = new Observable(observer => {
  let i = 0;
  const id = setInterval(() => observer.next(i++),
1000);
  return () => clearInterval(id);
});
```

“Hot” Observable

```
let i = 0;
let handlers= [];
setInterval(() => {
  handlers.forEach(fn => fn(i));
  i++;
}, 1000);

const hot = new Observable(observer => {
  const handler = (x) => observer.next(x);
  handlers.push();
  return () => {
    const index = handlers.indexOf(handler);
    if (index !== -1) {
      handlers.splice(index, 1);
    }
  };
});
```

“Hot” Observable

```
let i = 0;
let subject = new Subject();
setInterval(() => subject.next(i++), 1000);

const hot = new Observable(observer =>
  subject.subscribe(observer));
```

But now we don’t have teardown for the source!

“Hot” from “cold”

```
const cold = new Observable(observer => {
  let i = 0;
  const id = setInterval(() => observer.next(i), 1000);
  return () => clearInterval();
});

function makeHot(cold) {
  const subject = new Subject();
  let connectable = new Observable(observer => {
    return subject.subscribe(observer);
  });
  connectable.connect = () => cold.subscribe(subject);
  return connectable;
}

const hot = makeHot(cold);
```

“Hot” from “cold”

```
const cold = Observable.interval(1000);

function makeHot(cold) {
  const subject = new Subject();
  let connectable = new Observable(observer => {
    return subject.subscribe(observer);
  });
  connectable.connect = () => cold.subscribe(subject);
  return connectable;
}

const hot = makeHot(cold);
```

“Hot” from “cold”

```
const cold = Observable.interval(1000);

const hot = makeHot(cold).multicast(new Subject());
```

“Hot” from “cold”

```
const cold = Observable.interval(1000);

const hot = makeHot(cold).publish();
```

Recap

Cold

- subscription creates producer
- unicast

Hot

- subscription wraps external producer
- multicast
- Usually created from cold observables with `share`, `publish` or `multicast`

Subscription Management

Prevent resource leaks!

It’s important to manage your subscriptions carefully.

Unsubscribing is what tears down your data producers. Leaving a subscription running will likely result in memory and other resource leaks in your app!

Managing imperatively

```
const subscription = source.subscribe(observer);

// later

subscription.unsubscribe();
```

Managing imperatively

```
const subscription1 = source1.subscribe(observer);
const subscription2 = source2.subscribe(observer);
const subscription3 = source3.subscribe(observer);
const subscription4 = source4.subscribe(observer);
const subscription5 = source5.subscribe(observer);
const subscription6 = source6.subscribe(observer);
const subscription7 = source7.subscribe(observer);
const subscription8 = source8.subscribe(observer);



// later


subscription1.unsubscribe();
subscription2.unsubscribe();
subscription3.unsubscribe();
subscription5.unsubscribe();
subscription6.unsubscribe();
subscription7.unsubscribe();
subscription8.unsubscribe();


// oops?
```

Managing (mostly) Declaratively

```
const kill1 = Observable.fromEvent(button, ‘click’);
const kill2 = getStreamOfRouteChanges();
const kill3 = new Subject();

const merged = Observable.merge(
  source1.takeUntil(kill1),
  source2.takeUntil(kill2)
  source3.takeUntil(kill3);
);

const sub = merged.subscribe(observer);

// later

sub.unsubscribe();

// or any of the kill events could fire…
kill3.next(true);
```

Advantages to declarative approach

- Less likely to miss unsubscribing from a resource
- Can compose cancellation from any event source

Subscription rule of thumb:

If you find yourself managing more than one or two subscriptions you’re more likely to miss an unsubscribe.

Another approach to Subscription management

Let your framework or libraries handle it for you.

Use In Angular 2

(finally)

BYORX
(Bring your own RxJS)

Rather than include ALL of Rx.. (import Rx from ‘rxjs’)

…You can pull in just what you need, since RxJS 5 is modular.

This will reduce your deployed application size.

BYORX
(Bring your own RxJS)

- Add a file in your app root (I call mine `` `app.rx.ts` ``)
- Build your own Rx with RxJS 5 patch modules
  - export { Observable } from ‘rxjs/Observable’;
  - import ‘rxjs/add/operator/operatorName’;
  - import ‘rxjs/add/observable/fromWhatever’;

app.rx.ts

```
// direct exports
export { Observable } from 'rxjs/Observable';
export { Subject } from 'rxjs/Subject';

// static methods
import 'rxjs/add/observable/timer';
import 'rxjs/add/observable/empty';

// operators
import 'rxjs/add/operator/scan';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/switchMap';
import 'rxjs/add/operator/startWith';
import 'rxjs/add/operator/do';
```

Usage in a component

```
// import all Rx stuffs through your module
import { Observable, Subject } from ‘../
app.rx’;
```

Advantages to this approach

- Better than using patch operators at the top of every file
- Can point your Rx module at different implementations of Observable if you want
- Easier to figure out which operators you’re using in your app

Two Basic Subscription Methods

- OnInit, OnDestroy
- async pipe

OnInit, OnDestroy

```
class MyComponent implements OnInit, OnDestroy {
  subscription: Subscription;

  value: string;

  source$: Observable<number> = Observable.interval(1000);

  ngOnInit() {
    this.subscription = this.source$.subscribe(
      (value) => this.value = value
    );
  }

  ngOnDestroy() {
    if (this.subscription) this.subscription.unsubscribe();
  }
}
```

OnInit, OnDestroy

Pros

- Granular control
- Doesn’t have to be OnInit and OnDestroy

Cons

- Could end up maintaining too many Subscriptions
- Easier to miss an unsubscription, causing leaks
- More verbose

Async Pipe

```
@Component({
  …
  template: `<span>{{ value$ | async }}</span>`
})
export class MyComponent {
  value = Observable.interval(1000);
}
```

Async Pipe Gotcha

```
let counter = 0;

@Component({
  …
  template: `
    <span>{{ foo$ | async }} {{ bar$ | async }}</span>
  `
})
export class MyComponent {
  value$ = new Observable(observer => {
    if (counter++ > 1) throw new Error(‘one only!’);
    observer.next({ foo: ‘hi’, bar: ‘there’ });
    observer.complete();
  });

  get foo$() {
    return this.value$.map(x => x.foo);
  }

  get bar$() {
    return this.value$.map(x => x.bar);
  }
}
```

Async Pipe Gotcha Fix

```
let counter = 0;

@Component({
  …
  template: `
    <span>{{ foo$ | async }} {{ bar$ | async }}</span>
  `
})
export class MyComponent {
  value$ = new Observable(observer => {
    if (counter++ > 1) throw new Error(‘one only!’);
    observer.next({ foo: ‘hi’, bar: ‘there’ });
    observer.complete();
  })
  .share();

  get foo$() {
    return this.value$.map(x => x.foo);
  }

  get bar$() {
    return this.value$.map(x => x.bar);
  }
}
```

Async Pipe

Pros

- terse
- no subscription management

Cons

- subscription management limited to what is displayed
- Encourages too much use of `share()`