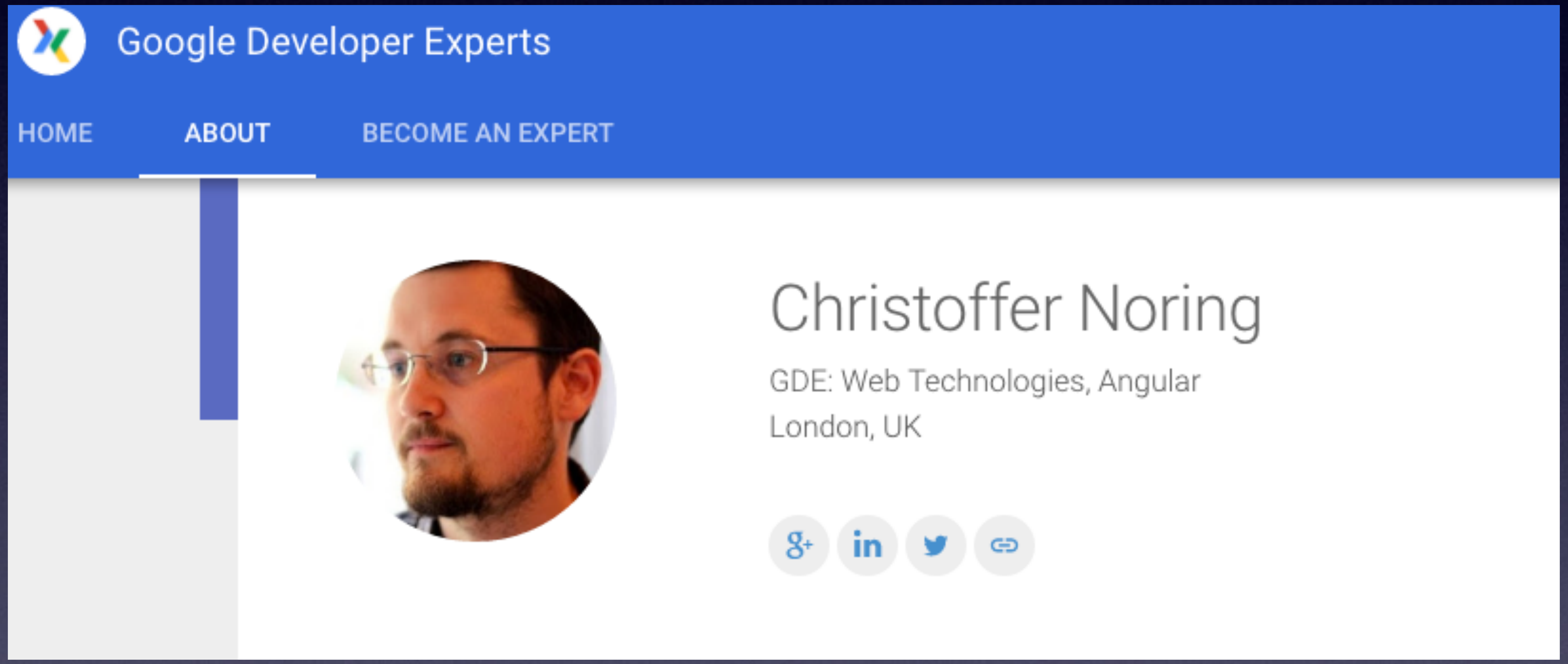
# RxJs  - demystified

*Lodash for async*

Christoffer Noring
@chris_noring

# Who am I

@chris_noring



Google Developer Experts

HOME ABOUT BECOME AN EXPERT

Christoffer Noring

GDE: Web Technologies, Angular

London, UK

# What is reactive programming?

Is a programming paradigm

Focuses on propagating changes
**without** us having to explicitly specify how the propagation happens

State **what** our code should to
**without** having to code every line

# Functional programming

Avoid mutable state
Avoid side-effects
**Functional composition** over **object composition**

= programs with predictable outcome

# Functional over imperative

Imperative

```
var list = [ 1,2,3,4,5 ];
list.forEach( function(value) {
   value += 1;
})
```

X number invocation changes state

Functional

```
var list = [ 1,2,3,4,5 ];
var newList = list.map( function(value) {
  return value + 1;
})
```

**list** NOT mutated/changed
but projected

Produces a new list

# Reactive programming

Async data streams

Turn
click **events**
user **inputs**
**data** from server
**callbacks** into a **stream**

One paradigm/ api to rule them all

Observable pattern
= Rx pattern
Iterative pattern

*Push-based behaviour,*
*don't call us, we'll call you*

**Observable**, *observable sequence*

Emits its values in order, like an iterator,
BUT pushes values as they become available

Has access to a producer in an observable pattern

Doesn't start streaming unless it has at least
one Observer subscribed to it

**Observer**

Consumer of an observable

Like listener in observable pattern

At the heart of everything is the

**Observable**

# Promise vs Array vs **Observable**

## Array

```
list
.map( x = > x.prop )
.filter( x => x > 2 )
.take( 2 )
```

## Promise

```
service.get()
.then( x => console.log(x)  )
.catch( err => console.log(err) )
```

## Observable

```
list
.map( x = > x.prop )
.filter( x => x > 2 )
.take( 2 )
.subscribe(
 x => console.log(x),
 err => console.log(err)  )
```

Array like,
handles async

but can also
- Cancelled
- Retried

# Observable vs Observable pattern

```
var stream = new Rx.Observable(function(observer) {
    observer.onNext('message');
})

stream.subscribe( function(val){
   console.log( val );
})
```

```
function Producer(){
    this.listeners = [];
}

Producer.prototype.add = function(listener){
    this.listeners.push( listener );
}

Producer.prototype.notify = function(message){
    this.listeners.forEach( function(listener){
        listener.update( message );
    })
}
```

## Observable is usually NOT the producer, it has access to one though

```
var stream = new Rx.Observable(function(observer){
   var producer = new Producer();
   producer.nextValue( function(val){
       observer.onNext( val );
   })
})

function Producer(){

}

Producer.prototype.nextValue = function(cb){
     setTimeout(function(){
         cb( 1 );
    },2000);
}
```

Producer created internally =
“cold observable”
Also responsible for emitting value to observer

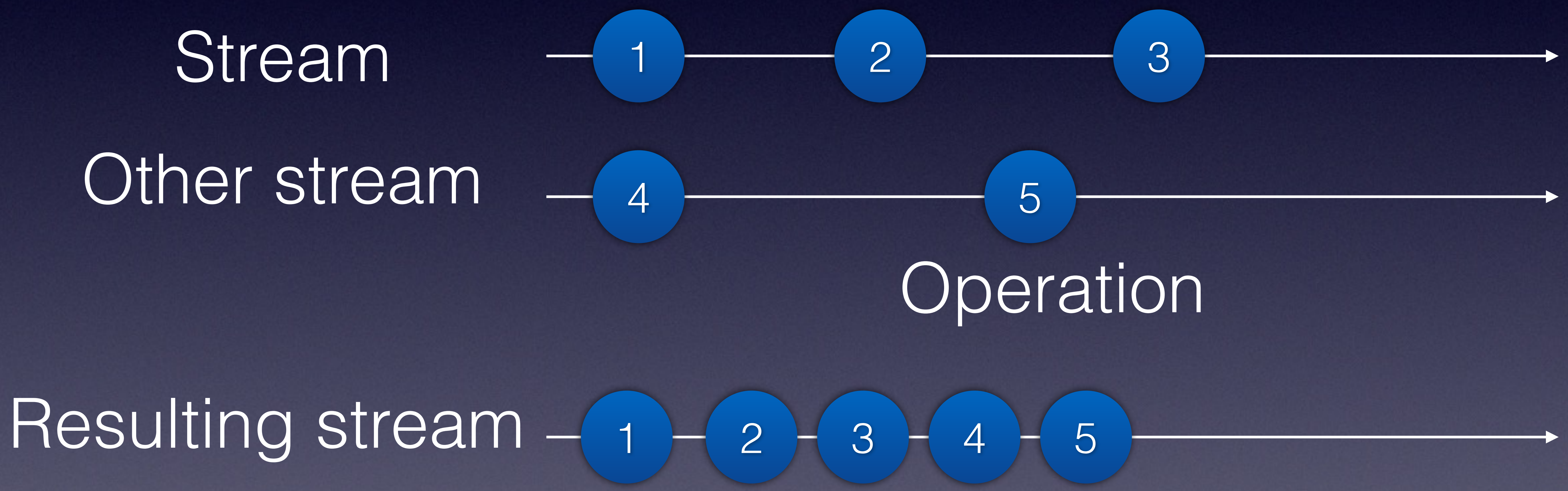
## Observable

```
Rx.Observable.create(
  onValueCallback,
  onErrorCallback,
  onCompletedCallback
)
```

Optional

## Marble diagrams

Most operators are covered at rxmarbles.com



# Observable

with error

```
var stream = Rx.Observable.create(function(observer){
   observer.onNext(1);
   observer.onNext(2);
   observer.onNext(3);
   observer.onError( ‘there is an error’ )
})

stream.subscribe(
   function(data){ console.log( data ); },
   function(error) { console.error( error ); }
)
```

# Observable

with completed

```
var stream = Rx.Observable.create(function(observer){
   observer.onNext( 1 );
   observer.onNext( 2 );
   observer.onNext( 3 );
   observer.onCompleted();
})

stream.subscribe(
   function(data){ console.log( data ); },
   function(error){ console.error( error ); },
   function(){ console.info( “completed” ); }
)
```

## Observable cancelling

```
var homemadeStream = Rx.Observable.create((observer) => {
    var i=0;

    var handle = setInterval(function() {
        observer.onNext( i++ );
    }, 500);

    return function(){
        console.log('Disposing timeout');
        clearTimeout( handle );
    }
});

var subscription2 = homemadeStream.subscribe((val) => {
    console.log('Homemade val',val);
});

setTimeout(function() {
    console.log('Cancelling homemadeStream');
    subscription2.dispose();
},  1500);
```

Produce values till someone calls **dispose**

Define whats to happen on **dispose**

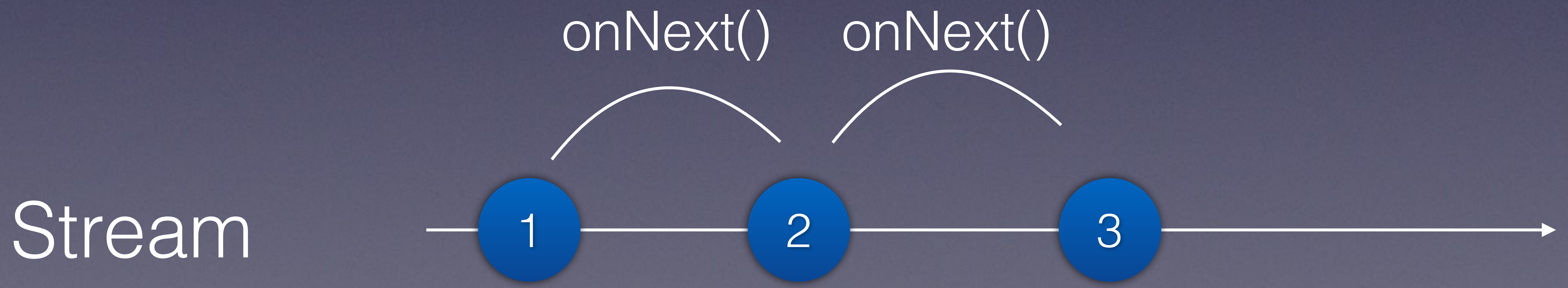
Calling **dispose**

# Observable

create your own

```
var stream = Rx.Observable.create(function(observer){
  observer.onNext(1);
  observer.onNext(2);
  observer.onNext(3);
})

stream.subscribe( function(data){ console.log( data ); } )
```



# Observable

## wrap ajax

```
var stream = Rx.Observable.create(function(observer){
  var request = new XMLHttpRequest();

  request.open( ‘GET’, ‘url’ );
  request.onload = function(){
    if(request.status === 200) {
      observer.onNext( request.response );
      observer.onCompleted();
    } else {
       observer.onError( new Error( request.statusText ) )
    }
  }

  request.onerror = function(){ observer.onError( new Error(‘unknown error’) );  }
  request.send();
})

stream.subscribe( function(result) { console.log( result ); } )
```

Producer

# Operators

120+ operators

Categories

Combination

Conditional

Filtering

Creational

Transformation

**Error handling**

Utility

Multicasting

## Operators

multicasting

# Operators

multicasting

**Cold** observable = producer is **not** shared

A producer is the source of values for your observable.

```
var cold = new Observable((observer) => {
  var producer = new Producer();
  // have observer listen to producer here
});
```

```
var producer = new Producer();
var hot = new Observable((observer) => {
  // have observer listen to producer here
});
```

## Operators

multicasting

Going from hot to cold

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

stream.subscribe(function(val){
    console.log('Subscriber 1', val);
});

stream.subscribe(function(val){
    console.log('Subscriber 2', val);
});
```

By definition cold,
as both subscriber start from 0,1 ..
NOT shared

```
Subscriber 1 ,0

Subscriber 2 ,0
```

Starts on same number

# Operators

## multicasting

How to share?

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

 stream.subscribe(function(val){
     console.log('Subscriber 1', val);
 });


 setTimeout(function() {
     stream.connect();
 }, 2000);

 setTimeout(function() {
     stream.subscribe(function(val){
         console.log('Started after 4 sec, Subscriber 2', val);
     });
 }, 4000);
```

Sits and wait til

## Observable,

creating

```
Rx.Observable.fromArray([ 1,2,3,4 ])
Rx.Observable.range(1,3)
Rx.Observable.interval(miliseconds)
Rx.Observable.fromEvent(element, ‘event’);
Rx.Observable.fromArray(eventEmitter, ‘data’, function(){})
Rx.Observable.fromNodeCallback(fs.createFile)
Rx.Observable.fromCallback(obj.callback)
Rx.Observable.fromPromise(promise)
Rx.Observable.fromIterable(function *() { yield 20 })
```

# Observable

common operators

```
var stream = Rx.Observable.of(1,2,3,4,5);

stream.map((val) => {
    return val + 1;
})
.filter((val) => {
    return val % 3 === 0;
})
```

Investigates every value

# Observable

reducer

```
var stream2 = Rx.Observable.of(1,2,3,4,5,6);
var subscription2 = stream2.reduce((acc, curr) => {
    return acc + curr;
});

subscription2.subscribe((val) => {
    console.log('Sum', val);
})
```

## Operators

utility

# Operators

utility

## Do

```
var stream = Rx.Observable.of(1,2,3,4,5);

var subscription = stream
.do(function(val){
    console.log('Current val', val);
})
.filter(function(val){
    return val % 2 === 0;
});

subscription.subscribe(function(val){
    console.log('Val',val);
})
```

Echos every value without changing it, used for **logging**

# Operators

utility

toPromise

Used to create a promise from an observable

```
var stream = Rx.Observable.of(1,2,3,4,5);

stream
.filter(function(val){
    return val % 2 === 0;
})
.toPromise()
.then( function(val){
    console.log('Promise', val);
})
```

```
4
```

Only latest value that fulfils filter is returned

# Operators

utility

## let

```
var stream = Rx.Observable.of(1,2,3,4,5);
```

```
stream
.let( observable =>
        observable
        .filter(function(x){
            return x % 2 > 0;
        }))
        .take(2)
.subscribe(function(val){
    console.log('let', val);
})
```

Let's us operate on the observable.
We **don't** need to return what we do like with a normal operator

```
stream
.filter(function(val){
    return val % 2 === 0;
})
```

# Operators

utility

Delay,
the whole sequence

```
stream
.delay(1000)
.subscribe( (val) => {
    var newTime = new Date().getTime();
    console.log('Delayed', val + " " + (newTime - time));
})
```

```
….. 1 second
      1
      2
      3
```

```
Rx.Observable.merge(
    Rx.Observable.of('Marco').delay(1000),
    Rx.Observable.of('Polo').delay(2000)
).subscribe((val) => {
    var newTime = new Date().getTime();
    console.log('Merged Delay', val + " " + (newTime - time));
})
```

```
….. 1 second
    Marco
….. 2 second
     Polo
```

## Operators

combination

# Operators

## combination

```
var stream = Rx.Observable.fromArray([1,2,3,4]);

var stream2 = Rx.Observable.fromArray([5,6,7,8]);
```

### concat

```
var concat = Rx.Observable.concat(
    stream,
    stream2
 );
```

1, 2, 3, 4, **5, 6, 7 ,8**

first stream emits all values
then remaining value

### merge

```
var merge = Rx.Observable.merge(
  stream,
  stream2
);
```

1, **5**, 2, **6**, 3, **7**, 4 ,**8**

streams take turn

There is difference so choose wisely

# Operators

combination

## combineLatest

```
 var stream = Rx.Observable.fromArray([1,2,3,4]);

var stream2 = Rx.Observable.fromArray([5,6,7,8]);

var combineLatest = Rx.Observable.combineLatest(
    stream,
    stream2
);

combineLatest.subscribe( function(val) {
    console.log( 'combine ', val )
})
```

```
[1,5], [2,6], [3,7], [4,8]
```

## Operators

conditional

# Operators

conditional

```
var stream =
Rx.Observable
.fromArray([1,2,3,4])
.every( function(val){
   return val % 2 === 0;
});
```

false

```
var evenStream =
Rx.Observable
.fromArray([2,4,8])
.every( function(val){
   return val % 2 === 0
});
```

true

Condition needs to be fulfilled on all values

## Operators

filtering

# Operators

## filtering

Ignores all generated mouse click events for 2 seconds

ex1

```
var debounceTime = Rx.Observable
.fromEvent(button,'click')
.debounce(2000);

debounceTime.subscribe( function(){
    console.log('mouse pressed');
})
```

waits x ms and returns latest emitted

ex2

```
var debouncedStream = Rx.Observable
.fromArray([1,2])
.debounce(25);

debouncedStream.subscribe( function(val){
    console.log('debounce stream', val );
});
```

returns 2

# Operators

## filtering

Generate numbers

```
var mousePressedTimer = Rx.Observable.interval(1000);
var mouseUp = Rx.Observable.fromEvent(button,'mouseup');
```

Break condition

```
mousePressedTimer
.takeUntil( mouseUp )
.subscribe( function(val){
    console.log('mouse up has NOT happened yet',val);
}, function(err){},
 function(){
     console.log('condition fulfilled');
 })
```

# Operators

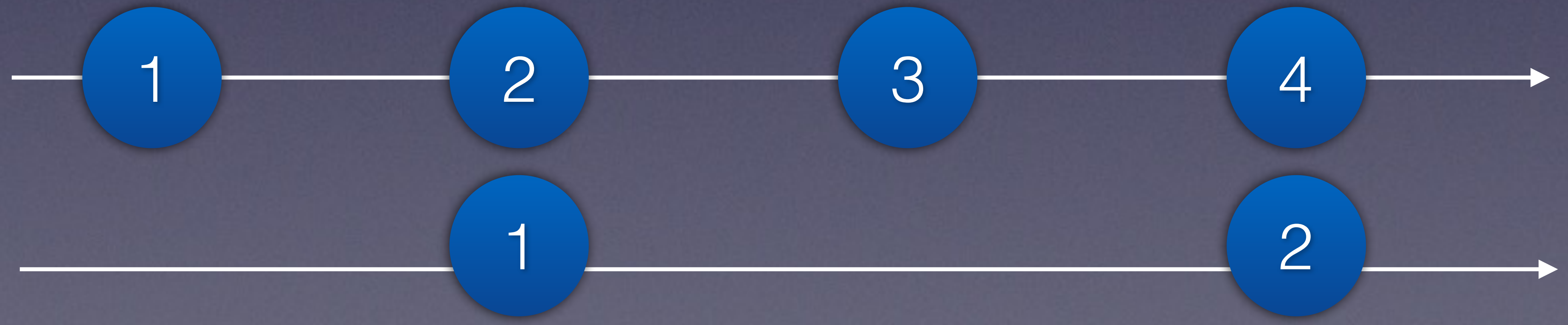
filtering

## Throttle

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

throttle.subscribe( function(val){
    console.log('Throttle', val );
});
```

Delay every value by x milliseconds



## Operators

transformation

# Operators

transformation

## Buffer

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

var bufferBy = Rx.Observable.fromEvent(document,'click');

var buffered = numbers.buffer( bufferBy );

buffered.subscribe(function(values){
    console.log('Buffered',values);
});
```

[1,2]
[3,4,5,6,7]
etc…

Numbers are generated to a buffer
**until** condition is met,
**then** the buffer content is emitted

# Operators

## transformation

### BufferTime

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


var bufferTime = numbers.bufferTime(2000);
bufferTime.subscribe(function(values){
    console.log("Buffer time",values);
})
```

Waits x miliseconds and then emits buffer

## Operators

transformation

### Expand

```
var expand = Rx.Observable
.of(2)
.expand(function(val){
    return Rx.Observable.of(1 + val);
})
.take(3);

expand.subscribe(function(val){
    console.log('Expand',val);
})
```

Recursively call provided callback
x times

## Operators

transformation

### Scan

Add accumulated and current

```
var scanning = Rx.Observable
.of({ prop : 'a'}, { prop2 : 'b'})
.scan((acc, curr) => Object.assign({}, acc, curr), {});

scanning.subscribe( function(val){
    console.log( val );
});
```

```
{ prop : 'a' }
{ prop : 'a', prop2 : 'b' }
```

```
var sum = Rx.Observable
.of(1,2,3)
.do(function(){
   console.log('Create side effects');
})
.scan(function(acc,curr) {
   console.log('Curr', curr);
   return acc + curr;
})
```

```
1
3
6
```

# Operators

transformation

Switch map,
complete something  based on a condition

```
breakCondition = Rx.Observable.fromEvent(document,'click');
breakCondition.switchMap( function(val){
    return Rx.Observable.interval(3000).mapTo('Do this if nothing breaks me');
})

breakCondition.subscribe( function(val){
    console.log('Switch map', val);
})
```

Intended action is completed/restarted by ‘breakCondition’

# Operators

transformation

Switch map

```
var timingBreakCondition = Rx.Observable.interval(3000);


var sourceTiming = timingBreakCondition.switchMap( function(val){
    return Rx.Observable.interval(1000).map(function(val){
        return 'Emitting' + (val + 1);
    })
})

sourceTiming.subscribe(function(val){
    console.log('Switch map timer', val);
})
```

Emits 1,2 until it is restarted by outer timer

# Operators

transformation

## Switch map

```
var breakCondition = Rx.Observable.fromEvent(document,'click');
var source = breakCondition.switchMap( function(val){
    return Rx.Observable.interval(3000).take(1)
    .flatMap(function(){
        return Rx.DOM.getJSON( 'data3.json' );
    });
})

source.subscribe( function(val){
    console.log('Switch map', val);
})
```

Get data every 3 second unless a ‘click’ happens

*Same example but do something useful like fetch data*

# Operators

transformation

flatMap

Takes an array of observables and merges these into one meta stream

```
var stream= Rx.Observable
    .fromArray([1,2,3,4])
    .take(2)
    .flatMap(function(val){
        return Rx.Observable.interval(500).map(function(){
            return val;
        });
    });
```

Creates observables

Will emit 1,2 in eternity

```
stream.subscribe(function(val){
    console.log( val );
});
```

# Operators

## transformation

### flatMap + json

1

```
var source = Rx.DOM.getJSON( 'data2.json' )
.flatMap( function(data) {
    return Rx.Observable.fromArray( data ).map(function(row){
        return row.props.name;
    });
} );
source.subscribe( function(data){
    console.log( data );
})
```

Takes array response and emits it row by row

2

```
.flatMap( function(data) {
    return Rx.Observable.fromArray( data ).pluck('props','name');
} );
```

3

```
.flatMap(Rx.Observable.fromArray).pluck('props','name')
```

# Operators

transformation

flatMap - example 2

```
flatmapExample = Rx.Observable.fromEvent(input,'keyup')
.map( function(ev){
    return ev.target.value;
})
.filter(function(text){
    return text.length >=3;
})
.distinctUntilChanged()
.flatMap( function(val){
    return Rx.DOM.getJSON( 'data3.json' );
})

flatmapExample.subscribe( function(result){
    console.log('Flatmap', result);
})
```

Transform event to char

Wait until we have 3 chars

Only perform search if this ‘search’ is unique

Excellent to use when coming from one stream to another

## Error scenarios

Capture error in .subscribe()

## Error handling

catch

```
var errStream = Rx.Observable.throw('Error');

var stream = Rx.Observable.create(function(observer){
    observer.onNext(1);
})

var merged = Rx.Observable
.merge( errStream, stream )


merged.subscribe( function(val){
    console.log('Val', val);
}, function(err){
    console.log('Err', err);
}, function(){
    console.log('completed');
})
```

Captured here but sequence interrupted,
completed NOT reached

## Error handling

Capture error in .subscribe() + completed stream

# Error handling

improved catch

Captured here but sequence interrupted,
completed IS reached

**stream** not processed though
can we improve it?

```
var errStream = Rx.Observable.throw('Error');

var stream = Rx.Observable.create(function(observer){
    observer.onNext(1);
})

var merged = Rx.Observable
.merge( errStream, stream )
.catch(function(err){
    return Rx.Observable.of(err);
});

merged.subscribe( function(val){
    console.log('Val', val);
}, function(err){
    console.log('Err', err);
}, function(){
    console.log('completed');
})
```

# Error handling

Wrap error stream before merging with other streams
so we don’t kill other valid streams

# Error handling

## ignoring

We need to handle the erroring stream better

From

```
var errStream = Rx.Observable
.throw('AAA thump..')
```

To

```
var errStream = Rx.Observable
.throw('AAA thump..')
.catch(function(err){
    return Rx.Observable.of(err);
});
```

This will emit **all** values and the wrapped error

## Error handling

Process all values and errors  by wrapping the errors, everything gets processed

# Error - ignoring

## other scenario

```
var errStream2 = Rx.Observable
.interval(200)
.take(3)
.select(function(val){
    if(val === 0) {
        return Rx.Observable.throw('Error stream');
    } else {
        return Rx.Observable.of(val);
    }
})
.select(function(observable){
    return observable.catch(Rx.Observable.return('Error handled'));
})
.selectMany( function(x){
    return x;
});
```

wrap

catch and rewrite

1,**1**,**2**

```
var merged = Rx.Observable
.merge( errStream2, stream )


merged.subscribe( function(val){
    console.log('Val', val);
}, function(err){
    console.log('Err', err);
}, function(){
    console.log('completed');
})
```

## Error handling

Set a policy for error handling of streams when merging so successful stream survive

# Error - ignoring

## other scenario

You have several sources and two terminates
You want the other normal source to work

```
var errStream = Rx.Observable
.throw('AAA thump..');

var errStreamWithFlattenedData = Rx.Observable
.interval(500)
.take(3)
.flatMap( function(val){
    if( val === 1 ) {
        return Rx.Observable.throw('crashing');
    }
    else {
        return Rx.Observable.return(val);
    }
})
```

```
var normalStream = Rx.Observable.return('anything');

var handledErrorStream =
Rx.Observable.onErrorResumeNext( errStream,
normalStream, errStreamWithFlattenedData );
handledErrorStream.subscribe(function(err){
    console.log('error stream ignored', err);
},
function(err){
    console.log("error stream ignored, error",err);
}, function(){
    console.log("completion of error stream ignored");
})
```

Dies midstream though,
you probably want to handle it fully

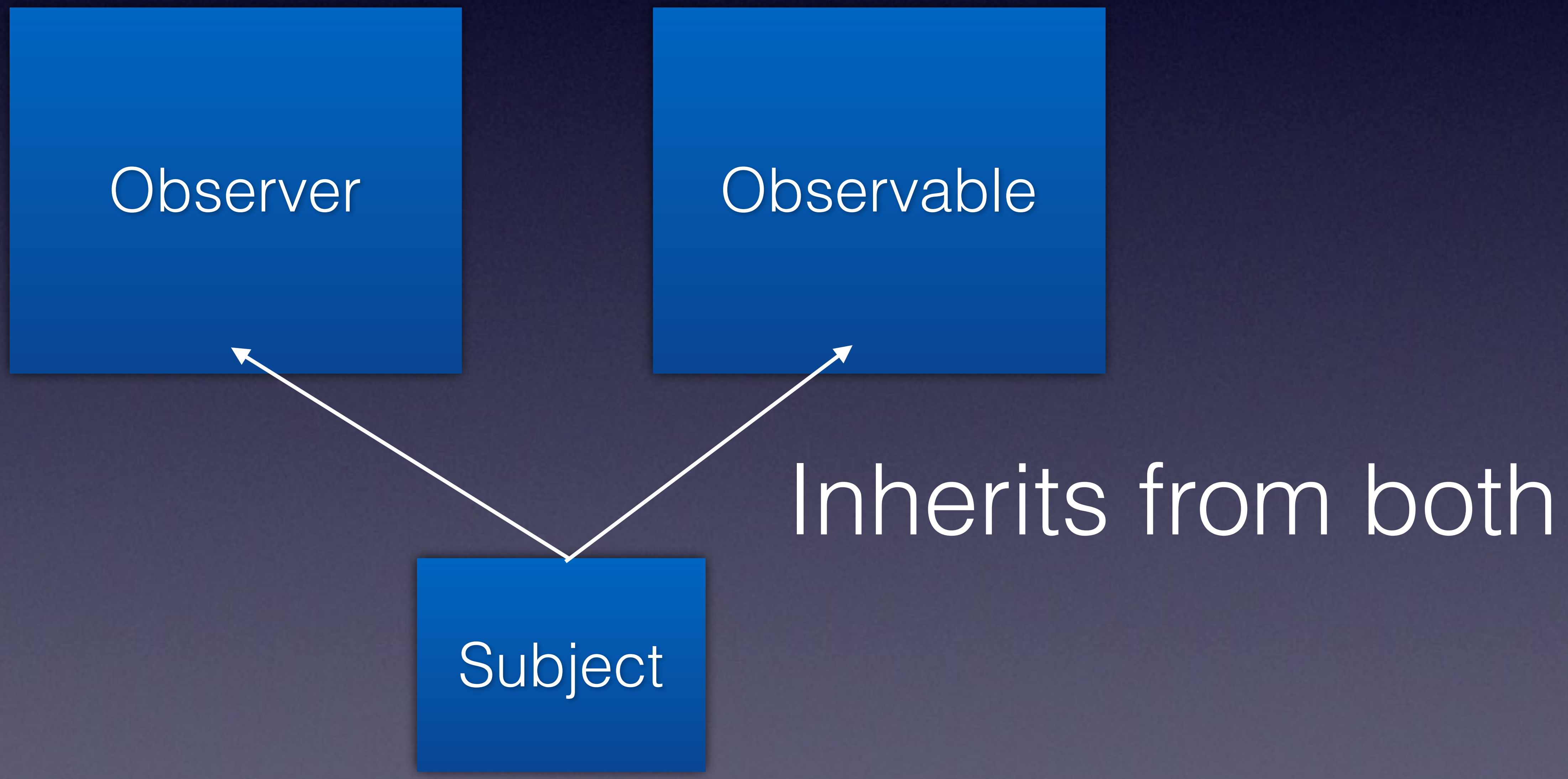
## Error - Retrying

```
var stream = Rx.DOM.get('/products.json').retry(5);

stream.subscribe((val) => {
    console.log('Data', val);
}, err => console.log(err));
```

Good thing with a shaky connection
5 attemps are made until it comes as an error

# Subject



*the subject can act as a proxy for a group of subscribers and a source*

# Subject
example

**Bold** is usually an operator like **.of()** or **.interval()**

Acts like an observable

```
var subject = new Rx.Subject();

subject.subscribe((val) => {
    console.log( 'Produced by subject', val );
});

subject.onNext(1);
subject.onCompleted();
```

Acts like a observer

```
function Producer(){
    this.i = 0;
}

Producer.prototype.nextValue = function(){
   return i++;
}

var observable = new Rx.Observable((observer) => {
    var producer = new Producer();

    observer.onNext(producer.nextValue());
    observer.onCompleted();
});

observable.subscribe( (val) => {
    console.log('Normal observable')
});
```

# Subject

```
var subject = new Rx.Subject();

var source = Rx.Observable.interval(500);

source.subscribe(subject);
```

Pass subject as an observer

```
subject.subscribe(
    (val) => {
        console.log('Sub', val);
    },
    (err) => console.log(err),
    () => console.log('completed')
);
```

Receives all the values pushed out by the source

```
setTimeout(function() {
    subject.onCompleted();
}, 3000);
```

Able to stop receiving values

## Subject

proxy one stream and add to it

```
var subject = new Rx.Subject();


var source = Rx.Observable.interval(500).take(3);

source.subscribe( subject );

subject.subscribe((val) => {
    console.log('Subject', val);
});

subject.onNext('Mess1');
subject.onNext('Mess2');



setTimeout(function() {
    subject.onCompleted();
}, 1600);
```

Listens to all values from **source**

Order important

Add to stream

## Subject

different types

### AsyncSubject

Returns a single value IF it completes
Cached forever

**Good for ajax requests**

### BehaviorSubject

Receives last emitted value
and then all subsequent ones

**Good for a placeholder
then real data to replace it scenario**

# Subject

different types

```
var subject = Rx.Subject();
subject.onNext(1);

subject.subscribe((val) = > {
    console.log('Replay', val);
})

subject.onNext(2);
subject.onNext(3);
```

Normal subject, everything before subscribe is lost

```
var subject = Rx.ReplaySubject();
subject.onNext(1);

subject.subscribe((val) = > {
    console.log('Replay', val);
})

subject.onNext(2);
subject.onNext(3);
```

Replay subject, nothing is lost

## Schedulers

bending time

## Schedulers

Is an operator async or sync?

Does it block the event loop?

Can I control how notifications are emitted?

*Give you fine grained control over how an observable emits notifications*

# Scheduler

bend it like beckham

Switch from Rx.Scheduler.**currentThread** to Rx.Scheduler.**default**

```
var stream = Rx.Observable.of(1,2,3,4,5);

var now = new Date().getTime();

stream.subscribe(
(val) => {
   console.log('Current thread')
},
err => {},
() => {
   var done = new Date().getTime();
   console.log( (done - now) + "ms" );
})
```

5ms

```
ar defaultStream = Rx.Observable.from([1,2,3,4,5],null,null,
Rx.Scheduler.default);
defaultStream.subscribe((val) => {
  console.log('Default', val);
},
err => {},
() => {
   var done = Date.now();
   console.log( "default " + (done - now) + "ms" );
})

console.log('default');
```

This happens first

**default** is async

**143ms !!**

# Scheduler

Use the right scheduler for the right things

| | |
|---|---|
| Constant Time Operations => | Rx.Scheduler.immediate |
| Tail Recursive Operations => | Rx.Scheduler.immediate |
| Iteration Operations => | Rx.Scheduler.currentThread |
| Time-based Operations => | Rx.Scheduler.default |
| Asynchronous Conversions => | Rx.Scheduler.default |
| Historical Data Operations => | Rx.HistoricalScheduler |
| Unit Testing => | Rx.TestScheduler |

Used right it can improve performance,
used wrong you will kill your app :)

## Schedulers

### testing

Because scheduler has its own virtual clock
Anything scheduled on that scheduler
will adhere to time denoted on the clock

I.e we can **bend time** for ex unit testing

# Schedulers

## testing

```
var onNext = Rx.ReactiveTest.onNext;
var scheduler = new Rx.TestScheduler();
var subject = scheduler.createColdObservable(
    onNext(100,'first'),
    onNext(200,'second')
);
```

What happens at what intervals

```
var result;

subject.subscribe((val) => {
    result = val;
});

scheduler.advanceBy( 100 );
```

Advance time

```
console.log('Should equal', result === 'first');
```

Assert

```
scheduler.advanceBy( 100 );
```

Advance time

```
console.log('Should equal', result === 'second');
```

Assert

# Schedulers

## testing

```
var testScheduler = new Rx.TestScheduler();

var stream = Rx.Observable.interval(1000, testScheduler)
.take(5)
.map((val) => {
    return val + 1
})
.filter((i) => {
    return i % 2 === 0
});
```

```
var result;
stream.subscribe((val) => result = val );

console.log('testing function');

testScheduler.advanceBy(1000);
testScheduler.advanceBy(1000);
testScheduler.advanceBy(1000);

console.log('Should equal', result === 2);

testScheduler.advanceBy(1000);
testScheduler.advanceBy(1000);
console.log('Should equal', result === 4);
```

We decide how fast time passes

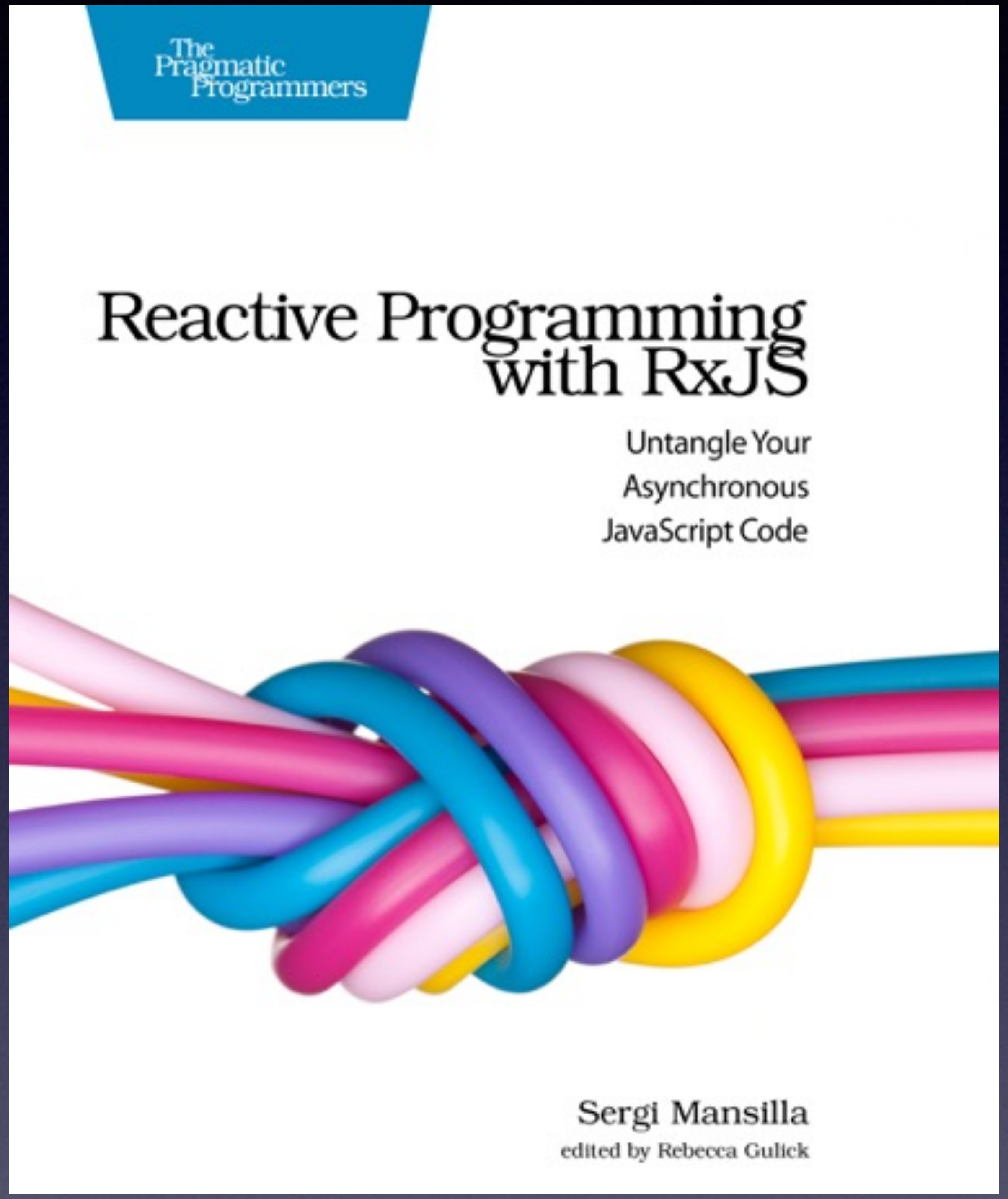
## Recipes

Buffer recipe, record interactions on UI
draw where the mouse has been

Replay a game, how a character have moved
left, right, jump, duck

Idea for a project, game where you get to play til you die
and take away move that kills you

# Further reading


The Pragmatic Programmers
Reactive Programming with RxJS
Untangle Your Asynchronous JavaScript Code
Sergi Mansilla
edited by Rebecca Gulick

https://xgrommx.github.io/rx-book

http://www.learnrxjs.io/

bacon.js

## Thank you