CRAFTSMEN

Reactive Programming

30 august 2018,

Daan Scheerens / Remco Weekers / Jonathan Oudshoorn

Agenda

16:00 - Introduction to RP / ReactiveX

16:45 - Workshop part I

17:30 - Diving deeper into ReactiveX

18:00 - FOOD! :)

18:45 - Workshop part II

19:50 - Wrap up

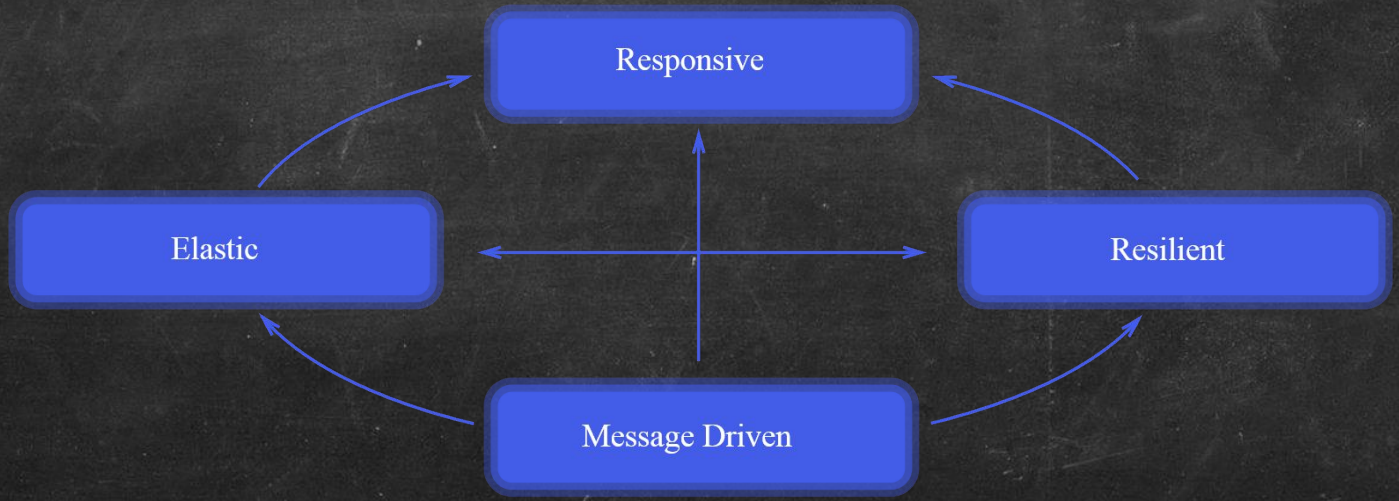
CRAFTSMEN

Why Reactive Programming?

- It is a hot topic
- Increased adoption in frameworks and libraries
- Will make *your* life easier
  - Also for you colleagues (...eventually)

Reactive Programming ⇏ Reactive Systems

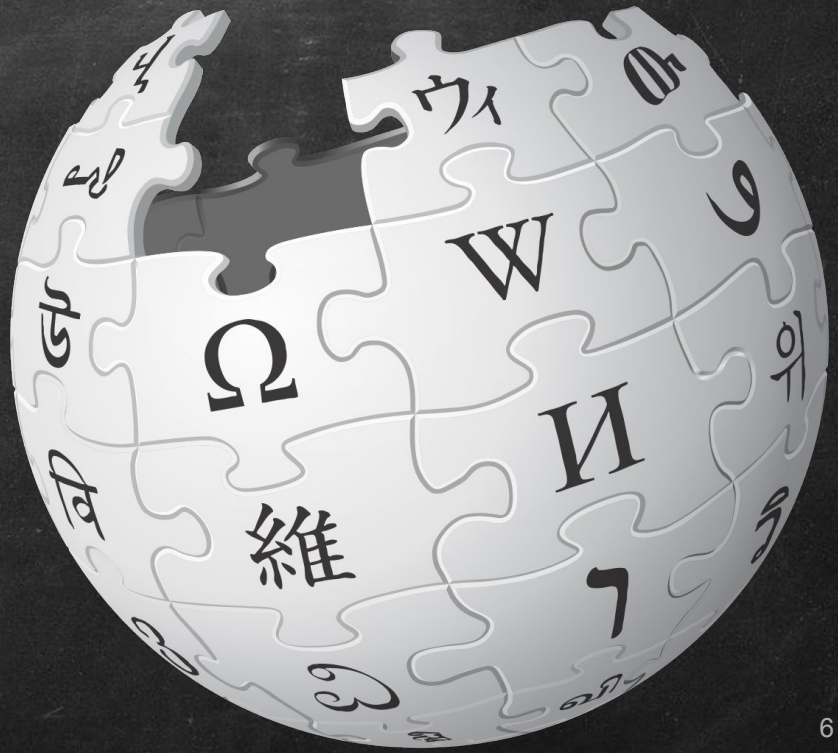
The reactive manifesto



CRAFTSMEN

Wikipedia

“In computing, reactive programming is a programming paradigm oriented around data flows and the propagation of change.”



CRAFTSMEN

Reactive Programming

- Programming paradigm focused on data flows
- Implementation level rather than architecture level
- Offers a convenient method for dealing with (asynchronous) data that:
  - **Becomes available** *(at some point in time)*
  - **Changes** *(over time)*
- Advantages
  - Makes asynchronous events more explicit
  - No callback hell
  - Clean and concise code
  - Focus on solving the task at hand

CRAFTSMEN

Propagation of change

```
let x = 3;

let y = 4;

let z = x * y;

console.log(z); // 12

y = 3;

console.log(z); // 12
```

```
let x$ = 3;

let y$ = 4;

let z$ = x$ * y$;

console.log(z$); // 12

y$ = 3;

console.log(z$); // 9
```

CRAFTSMEN

Reactive Programming libraries

- ReactiveX
- Akka reactive streams
- Project Reactor
- Bacon.js
- *Vert.x*
- ...
- ~~*React*~~

CRAFTSMEN

Why ReactiveX?

- Very popular: widespread adoption in many frameworks/libraries
- Native support by:
  - Angular 2
  - Spring 5
  - Vert.x
  - ...
- Bridges for other frameworks/libraries:
  - AngularJS
  - React
  - Ember
  - ...

CRAFTSMEN

ReactiveX History

- Originally developed @ Microsoft as RP framework for .NET
- Name stands for: ***Reactive Extensions***
- Open sourced in 2012
- Reimplemented in many other programming languages

Microsoft

CRAFTSMEN

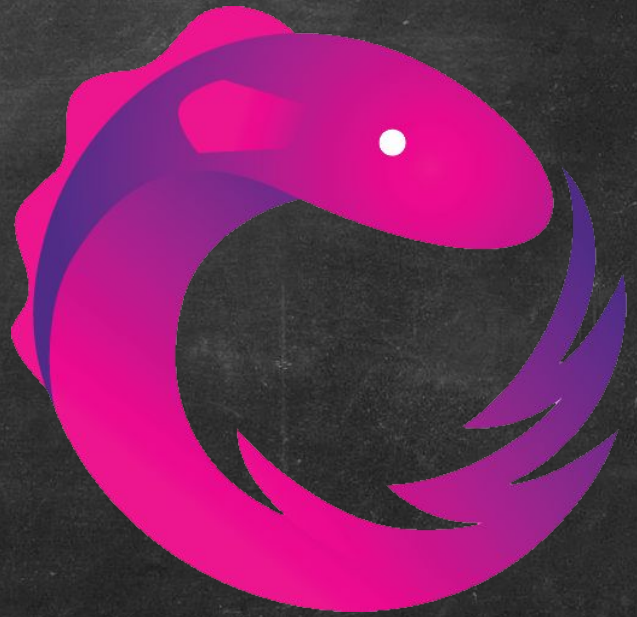
ReactiveX family overview

Java:	**RxJava**	Ruby:	**Rx.rb**
JavaScript:	**RxJS**	Python:	**RxPY**
C#:	**Rx.NET**	Groovy:	**RxGroovy**
C#(Unity):	**UniRx**	JRuby:	**RxJRuby**
Scala:	**RxScala**	Kotlin:	**RxKotlin**
Clojure:	**RxClojure**	Swift:	**RxSwift**
C++:	**RxCpp**	PHP:	**RxPHP**
Lua:	**RxLua**		

ReactiveX

“ReactiveX is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming”

http://reactivex.io/



CRAFTSMEN

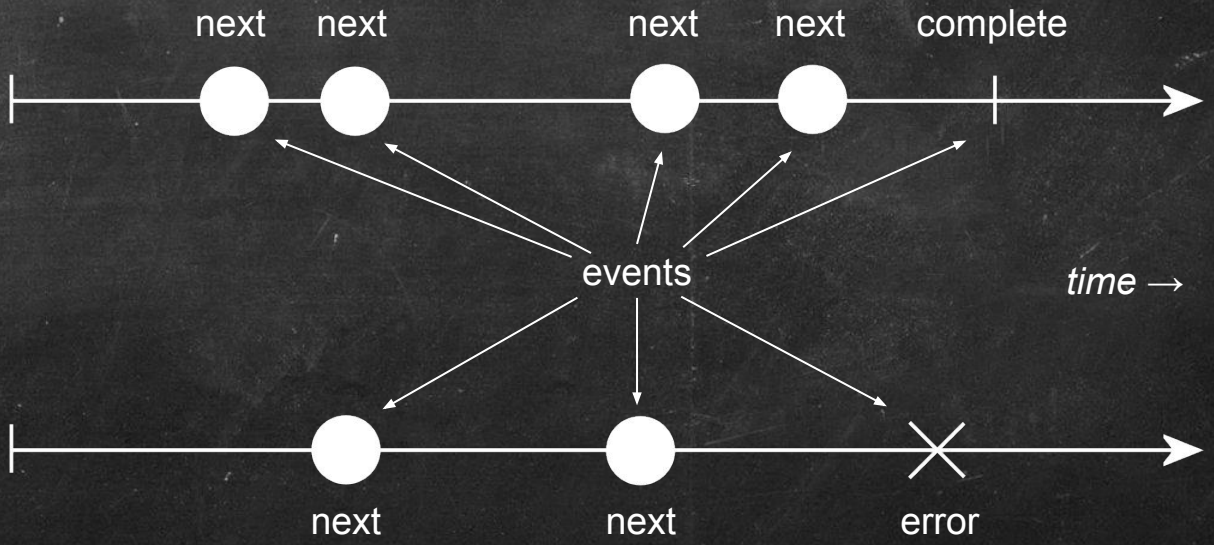
The observable:

Abstract stream of (a)synchronous data

Events:

- next (data)
- error (error)
- complete

Subscribe to observable
to receive events!



CRAFTSMEN

The observable “laws”

- May emit multiple values over time (next event)
- May emit nothing at all
- Either terminates with a complete event, an error event, *or does not terminate at all**

  ** as long as the application is running*
- No new events after termination
- Can be synchronous or asynchronous
- Can have multiple subscribers

CRAFTSMEN

Subscribing to observables

```
const message$ = giveMeData();

message$.subscribe(
    (value) => console.log('next: ' + value),
    (error) => console.log('error: ' + error),
    ()      => console.log('complete')
);
```

CRAFTSMEN

Subscribing with observer objects

```
const tweets$ = fetchTweets();

const observer = {
  next:     (value) => console.log('next: ' + value),
  error:    (error) => console.log('error: ' + error),
  complete: ()      => console.log('complete')
};

tweets$.subscribe(observer);
```

CRAFTSMEN

Unsubscribing from observables

```
const dataSubscription = data$.subscribe(observer);

dataSubscription.unsubscribe();
```

CRAFTSMEN

Relation with Observable pattern

Similarities

- Both are push based
- Both require subscription / unsubscription*

  ** unsubscription may happen automatically for observable streams*

Differences

- Observable streams are separated from their source
- Unsubscribing from observable streams goes through `Subscription` objects
- Observable streams can be composed
- Observable streams have separate error and complete channels

CRAFTSMEN

Relation with Iterator pattern

Similarities

- Both have an indefinite length

Differences

- Observable streams are push based instead of pull based
  and can therefore be asynchronous
- Observable streams can be composed
- Observable streams have separate error and complete channels

CRAFTSMEN

Relation with Java 8 streams

Similarities

- Both can be composed

Differences

- Observable streams can be asynchronous, Java 8 streams cannot
- Observable streams have separate error and complete channels

CRAFTSMEN

Relation with JavaScript (ES6) Promise

Similarities

- Both can be asynchronous
- Both have a separate error channel

Differences

- Observable streams support multiple values (over time)
- Observable streams don’t store history*, Promises do

  ** not by default at least*
- Observable streams have more options for composition
- You can’t unsubscribe from promises

Composing observable streams: operators

CRAFTSMEN

RxJS 6 operator flavours

Pipeable operators

```
observableStream$.pipe(
   operator1,
   operator2,
   operator3,
   //...
   operatorN
);
```

Static operators

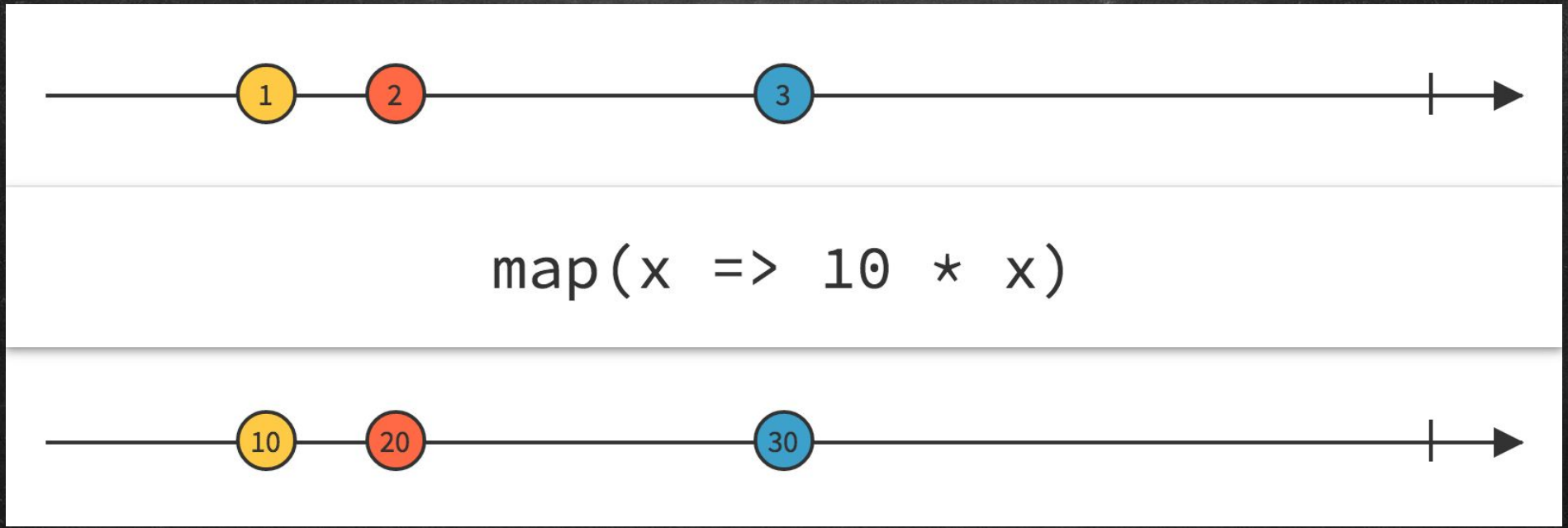
```
operator(
   observableStream1$,
   observableStream2$,
   observableStream3$,
   //...
   observableStreamN$,
);
```

CRAFTSMEN

Changes in RxJS 6

- No more patch operators
- Pipeable operators that also have a static counterpart are **deprecated**, e.g.:
  - `combineLatest, concat, merge, zip`
- Result selector parameter is **deprecated**
  - `map` operator should be used instead

map

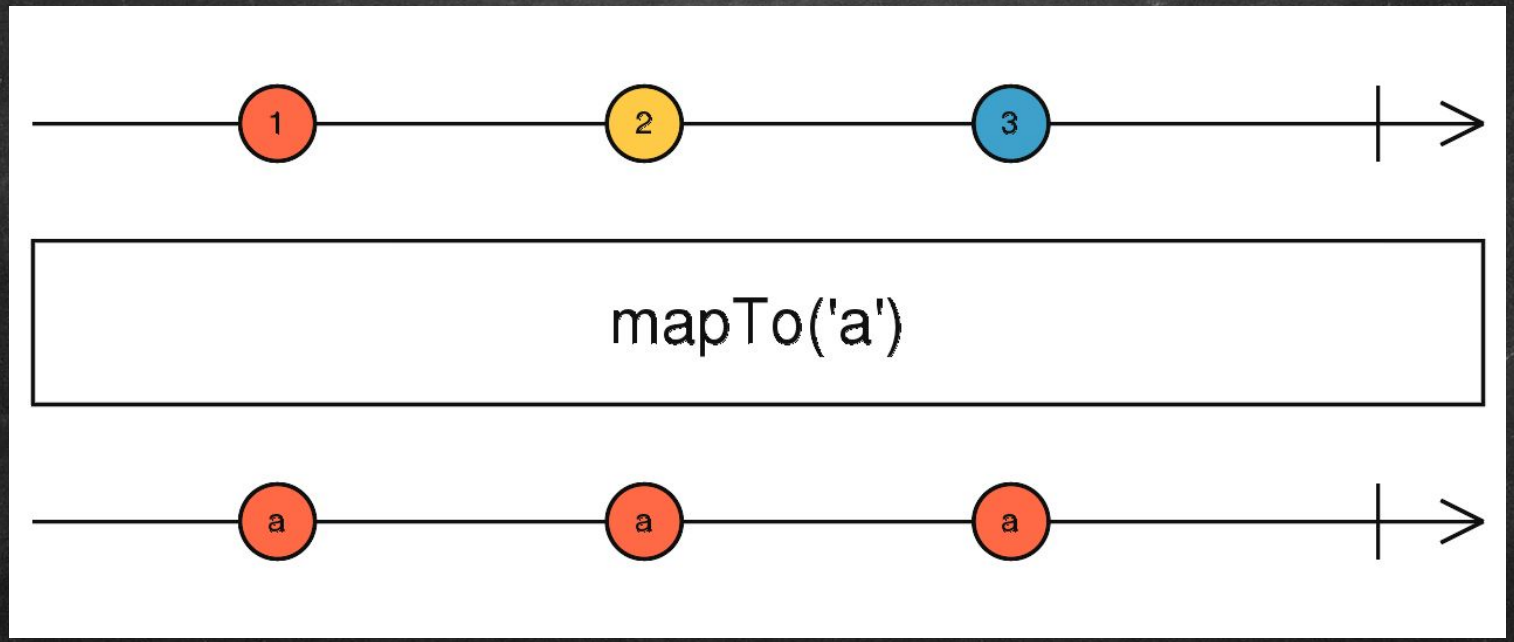


CRAFTSMEN

map

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

const number$ = of(1, 2, 3, 5, 8);

number$.pipe(
    map((x) => x * 10)
);
```

mapTo



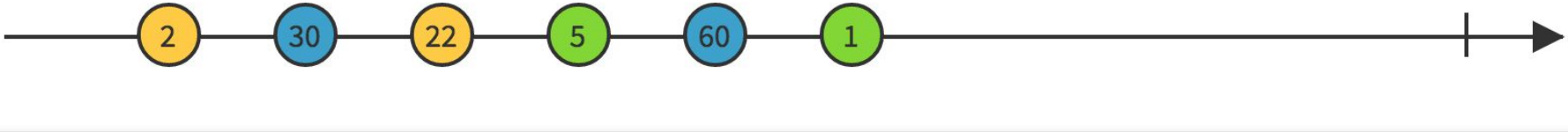
CRAFTSMEN

mapTo

```
import { of } from 'rxjs';
import { mapTo } from 'rxjs/operators';

const number$ = of(1, 2, 3, 5, 8);

number$.pipe(
    mapTo('a number happened somewhere, sometime...')
    // same as: map(() => 'a number happened somewhere, sometime...')
);
```

CRAFTSMEN

filter

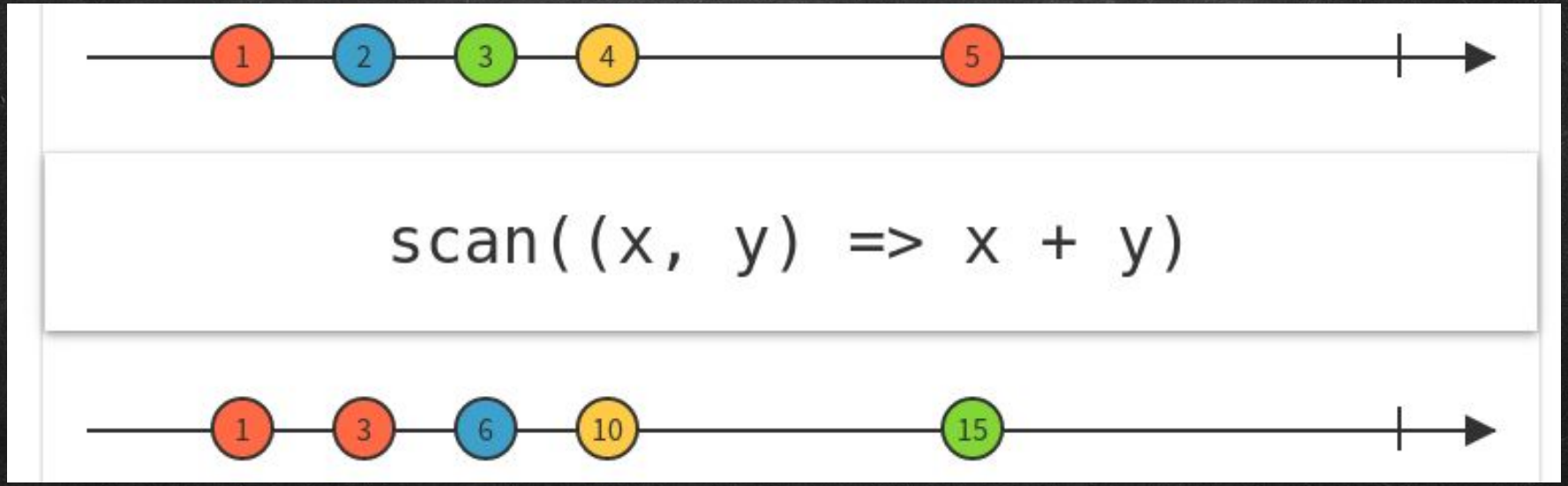



```
filter(x => x > 10)
```

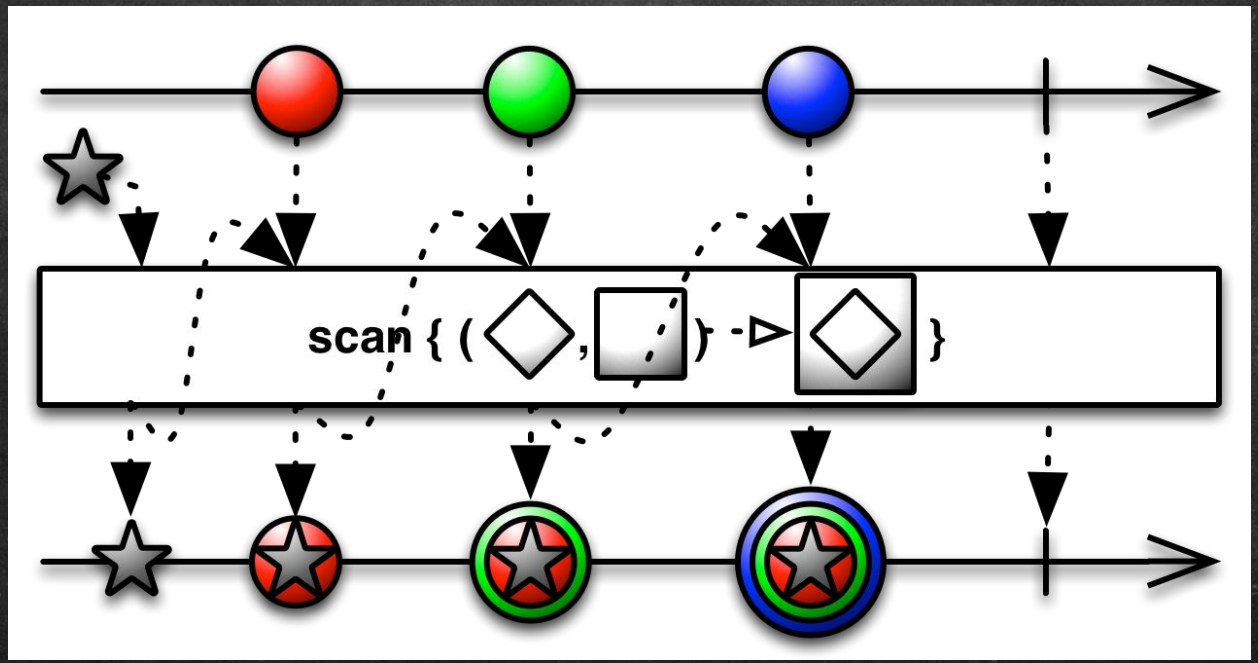
CRAFTSMEN

filter

```
import { of } from 'rxjs';
import { filter } from 'rxjs/operators';

const number$ = of(1, 2, 3, 5, 8);

number$
    .pipe(filter((x) => x % 2 === 0))
    .subscribe(console.log);
```


1
2
3
4
5
scan((x, y) => x + y)
1
3
6
10
15

`scan` *(with seed value)*



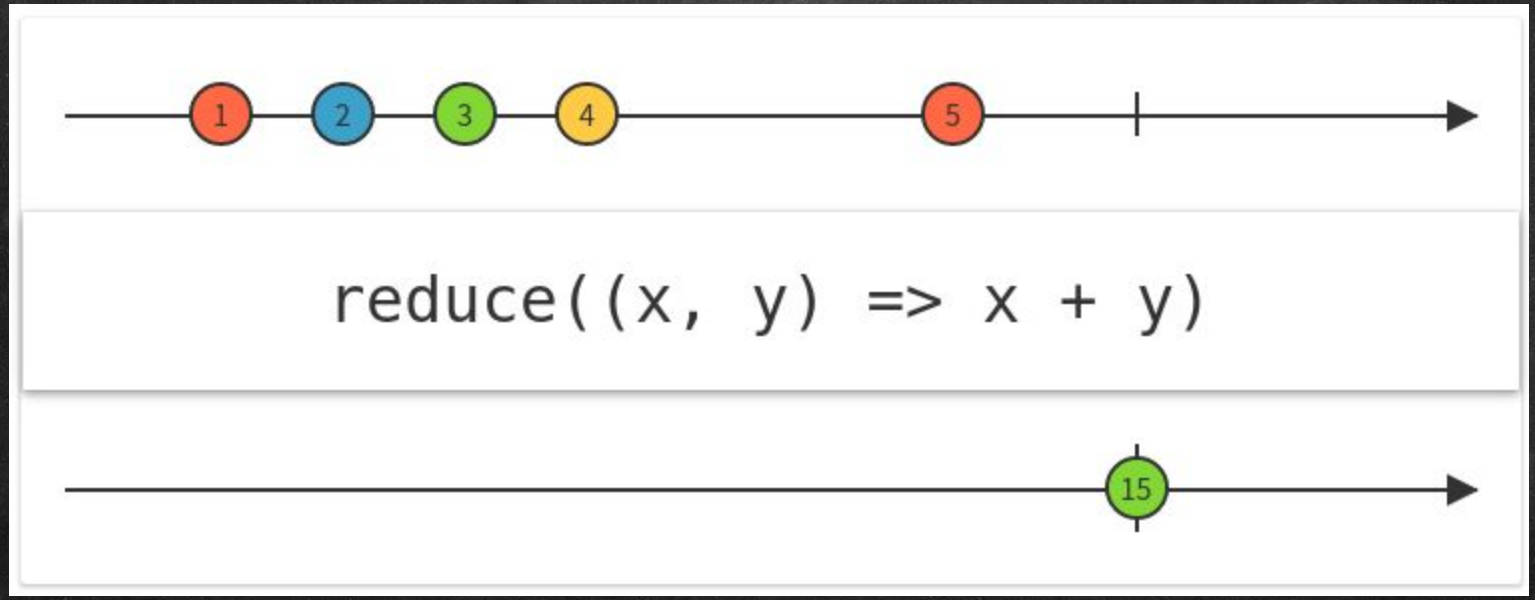
CRAFTSMEN

scan

```
import { of } from 'rxjs';
import { scan } from 'rxjs/operators';

const number$ = of(1, 2, 3, 5, 8);

number$.pipe(
    scan((x, y) => x + y) // Without seed value
);

number$.pipe(
    scan((x, y) => x + '[' + y + ']', 'values: ') // With seed value
);
```

seed value

reduce



CRAFTSMEN

reduce

```
import { of } from 'rxjs';
import { reduce } from 'rxjs/operators';

const number$ = of(1, 2, 3, 5, 8);

number$.pipe(
   reduce((x, y) => x + y) // Without seed value
);

number$.pipe(
   reduce((x, y) => x + '[' + y + ']', 'values: ') // With seed value
);
```

seed value

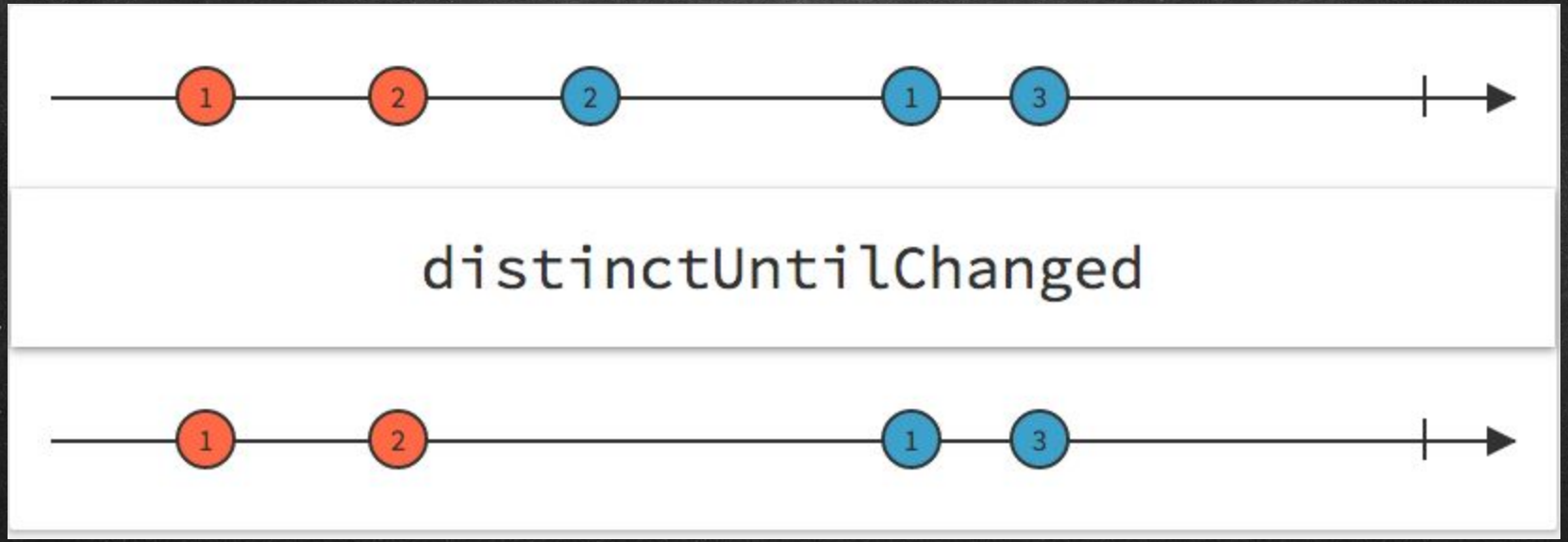
CRAFTSMEN

Typescript vs `scan` and `reduce`

There is a type inference issue when using scan or reduce with a seed value with a different type than that of the values of the input stream!

- Still an open issue on RxJS 6
- Workaround: provide explicit type arguments, e.g.:
  - `scan<number, string>((x, y) => x + y, '')`
- The workshop provides **patched** versions

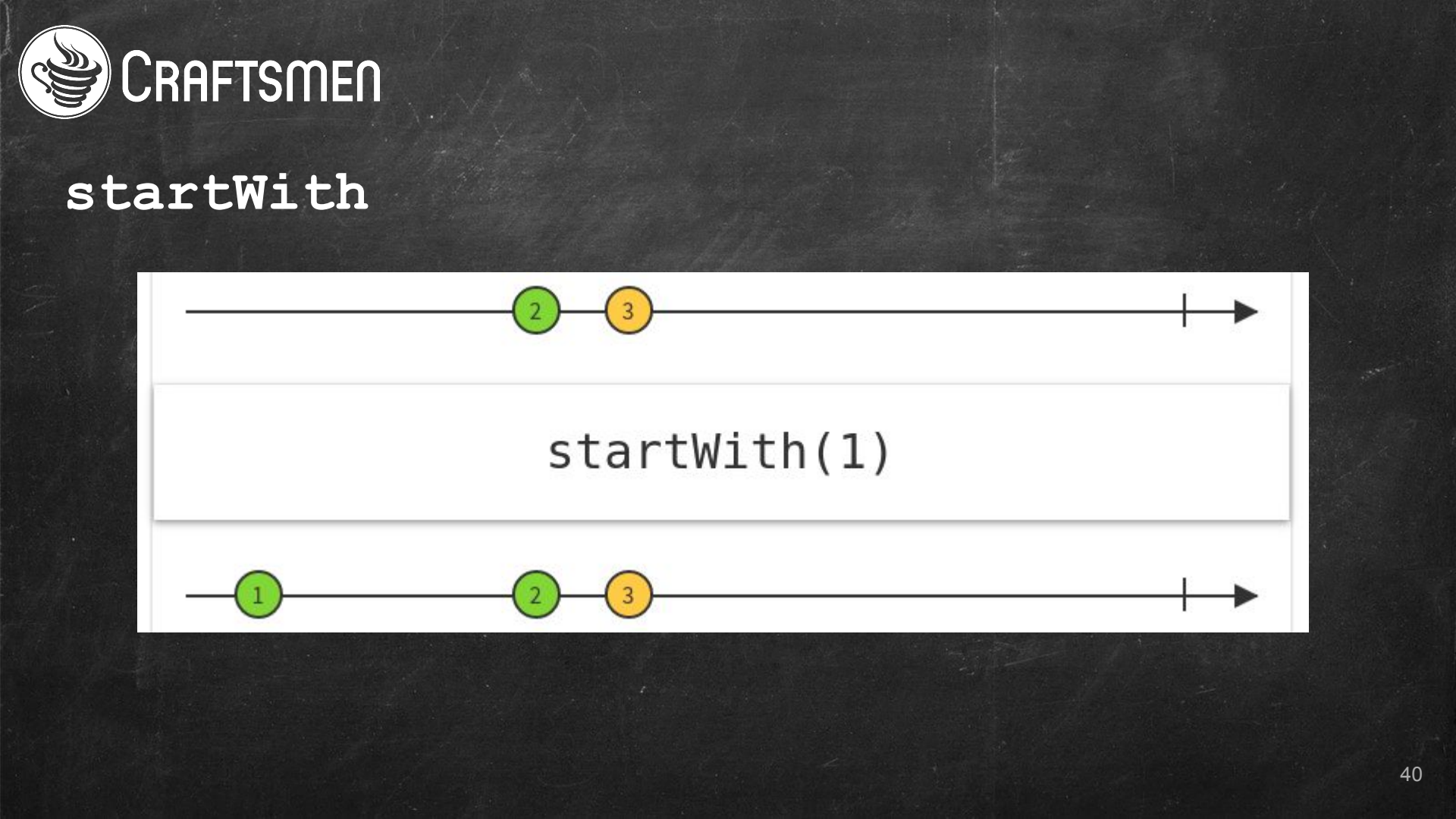
distinctUntilChanged



CRAFTSMEN

distinctUntilChanged

```
import { of } from 'rxjs';
import { distinctUntilChanged } from 'rxjs/operators';

const number$ = of(1, 2, 2, 3, 2, 3, 3, 3, 1);

number$
    .pipe(distinctUntilChanged())
    .subscribe(console.log);
```



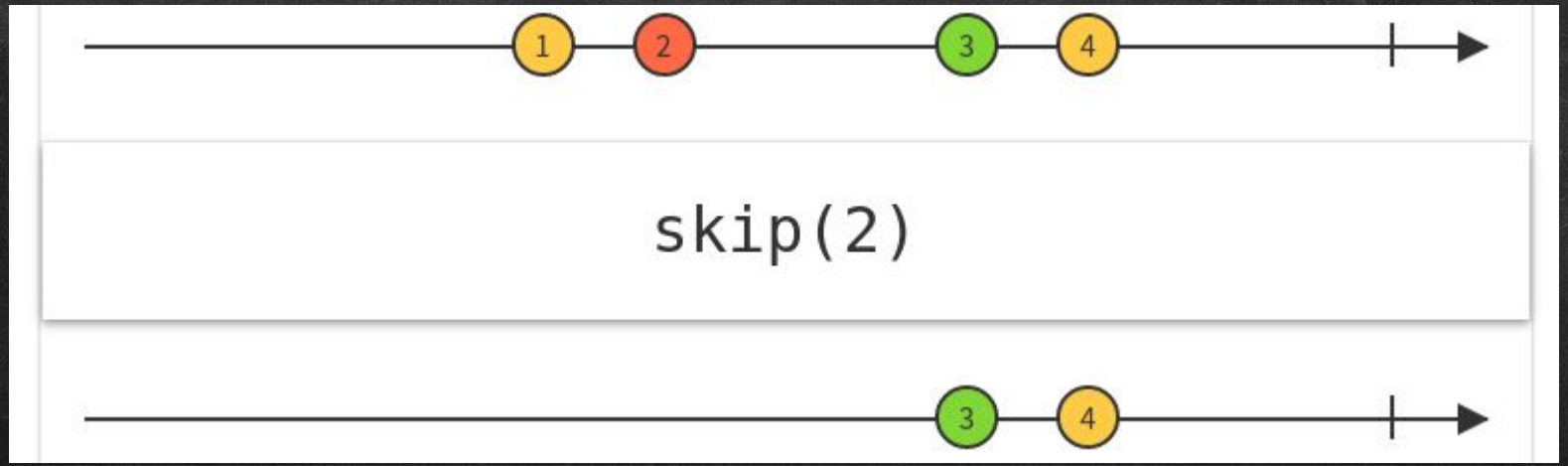
startWith

CRAFTSMEN

startWith

```
import { of } from 'rxjs';
import { startWith } from 'rxjs/operators';

const number$ = of(1, 2, 3, 4);

number$
    .pipe(startWith(-2, -1, 0))
    .subscribe(console.log);
```

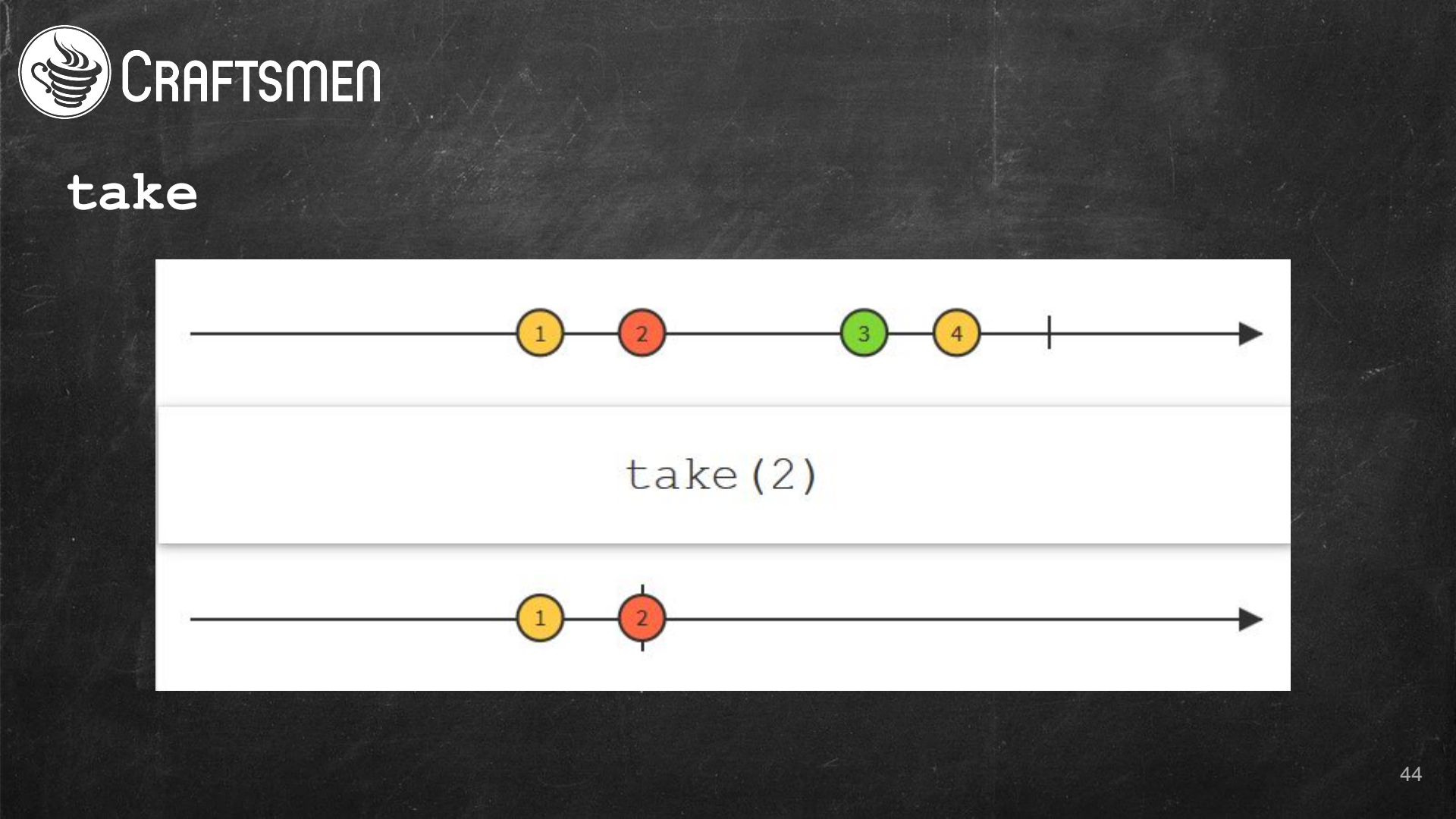
skip



CRAFTSMEN

skip

```
import { of } from 'rxjs';
import { skip } from 'rxjs/operators';

const number$ = of(1, 2, 3, 4);

number$
    .pipe(skip(2))
    .subscribe(console.log);
```


CRAFTSMEN
take
1
2
3
4
take(2)
1
2

CRAFTSMEN

take

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

const number$ = of(1, 2, 3, 4);

number$
    .pipe(take(2))
    .subscribe(console.log);
```

Live Demo!

CRAFTSMEN

Important characteristics

- Observables are immutable
  - Applying an operator returns **new** observable
- Observables are inert
  - Data only starts *“flowing”* by subscribing!
- Observable streams are not shared *(at least not by default)*
  - Each subscriber will receive its own flow
  - Operators will transform data for each individual subscriber

CRAFTSMEN

Quiz - What is the output?

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

const number$ = of(1, 2, 3, 4);

number$.pipe(
  map((x) => {
      const result = x * 2;
      console.log(result);
      return result;
  })
);
```

CRAFTSMEN

Quiz - What is the output?

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

const number$ = of(1, 2, 3, 4);

number$.pipe(
   map((x) => x * 2)
);

number$.subscribe(console.log);
```

CRAFTSMEN

Quiz - What is the output?

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

const a$ = of(1, 2, 3, 4);

let counter = 0;
const b$ = a$.pipe(
   map((x) => { counter++; return x * 2; })
);

a$.subscribe();
console.log(counter);
```

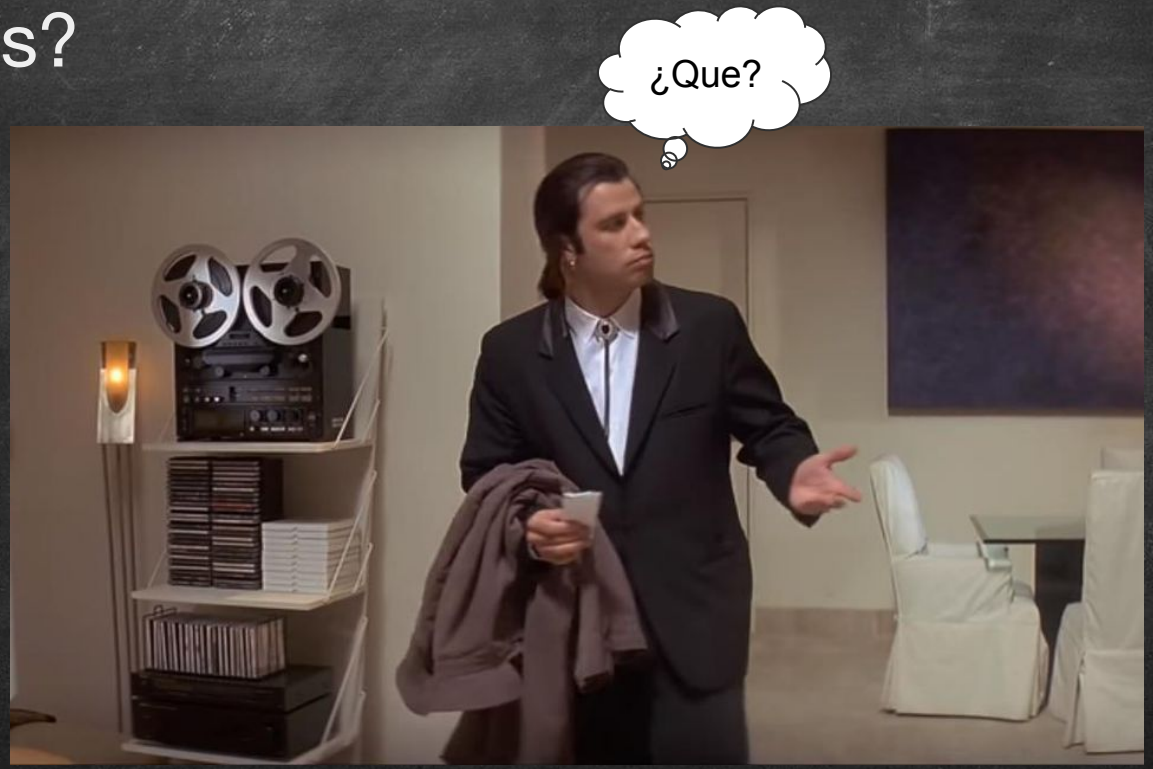
CRAFTSMEN

Quiz - What is the output?

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

const a$ = of(1, 2, 3, 4);

let counter = 0;
const b$ = a$.pipe(
   map((x) => { counter++; return x * 2; })
);

b$.subscribe(); b$.subscribe();
console.log(counter);
```

CRAFTSMEN

Questions?



CRAFTSMEN

Time to start coding!

- Clone/download the workshop repository
- Follow installation instructions in README.md
- Do workshop exercises **1** to **11**

Creating observables

Creation can be done in several ways:

- Static functions
- Using Subjects
- `new Observable()`

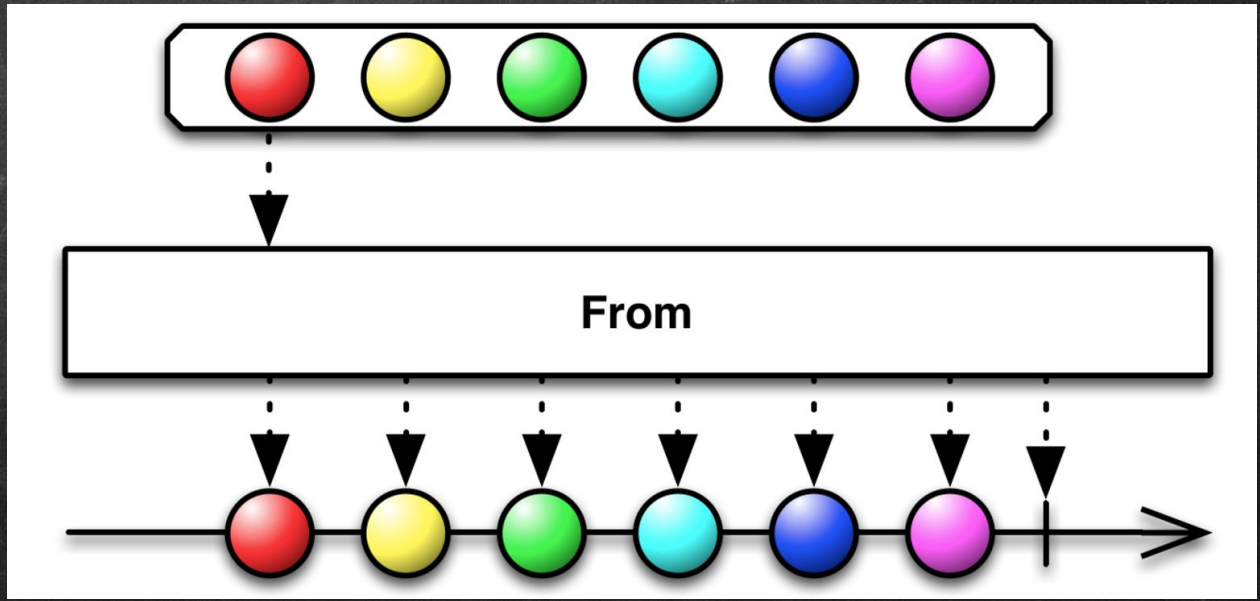
Creating observables - Static functions

Creation can be done with several methods:

- `range()`
- `interval()`
- `of()` / `from()`
- `...`

Creating observables - `of` / `from`

Converts a sequence of values to an observable



CRAFTSMEN

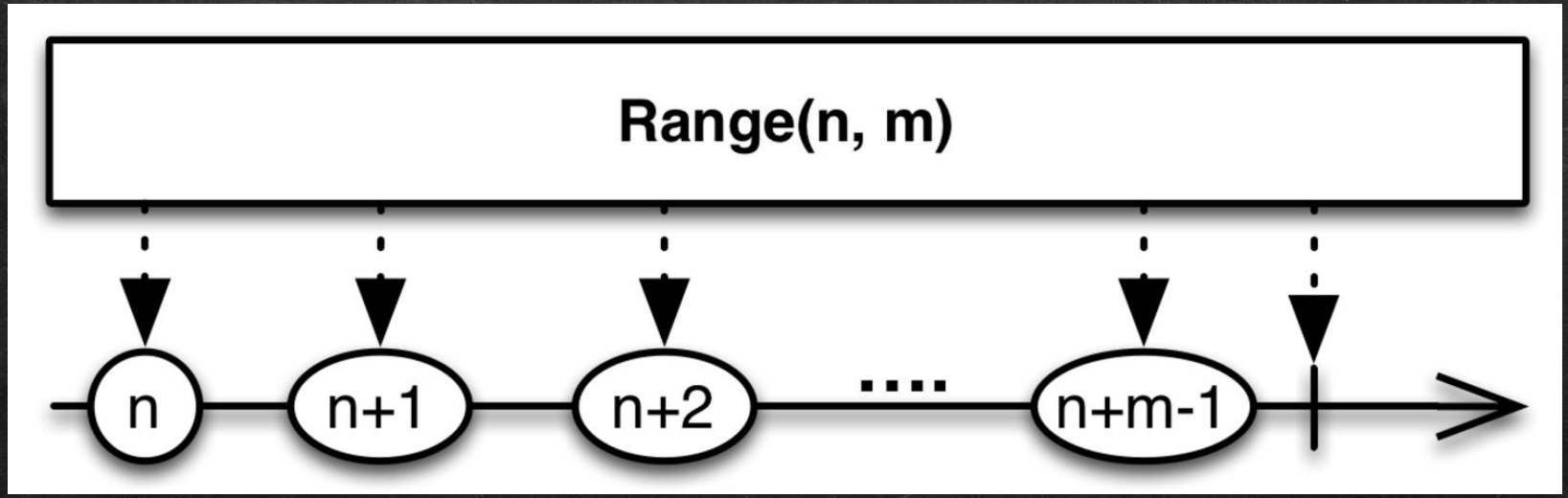
Creating observables - `of` / `from`

```
import { of, from } from 'rxjs';

const number$ = of(1, 2, 3, 4);

const words$ = from(['foo', 'bar', 'baz']);
```

CRAFTSMEN

Creating observables - `range`

Emits a sequence of increasing numbers within the specified range



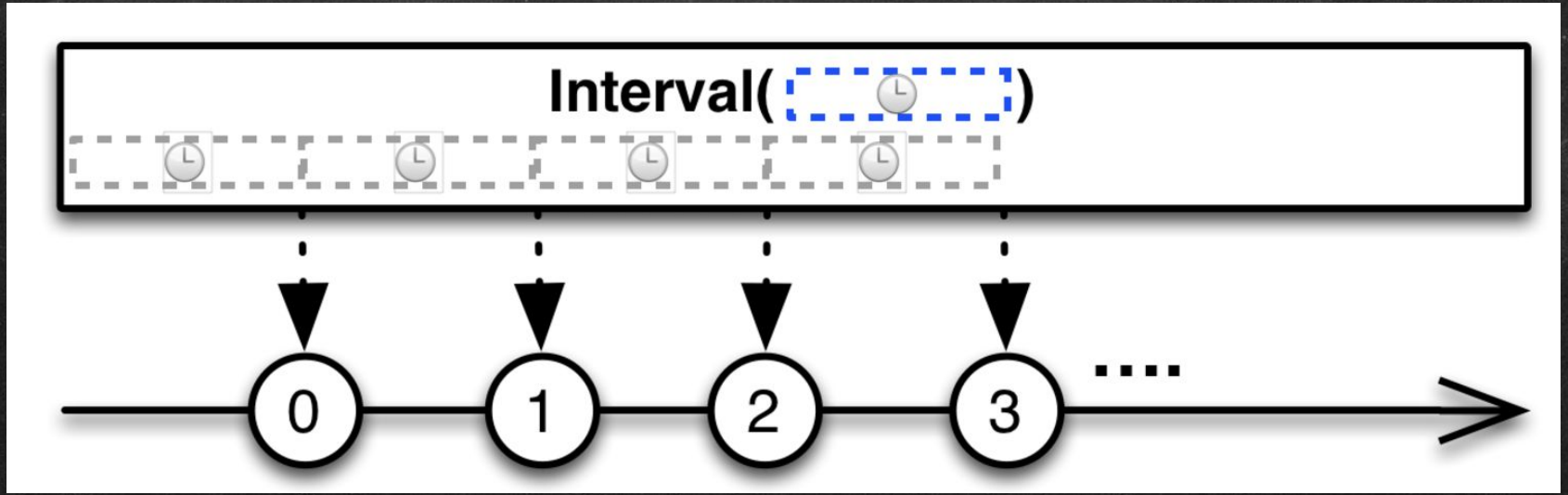
CRAFTSMEN

Creating observables - `range`

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

const theAwesomeNumbers$ = range(42, 1337);
```

Creating observables - `interval`

Emits a sequence of integers spaced by a time interval



Creating observables - `interval`

```
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';

interval(250)
    .pipe(take(10))
    .subscribe(console.log);
```

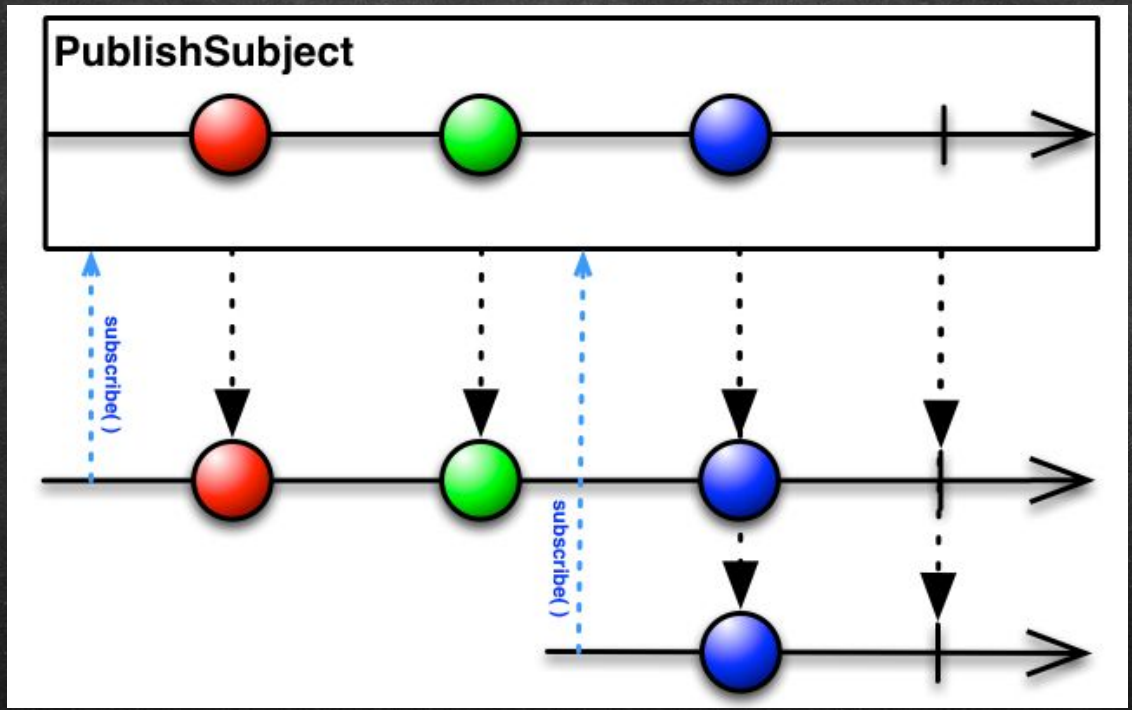
Creating observables - Subjects

Several implementations available:

- `PublishSubject`
- `ReplaySubject`
- `BehaviorSubject`
- `AsyncSubject`

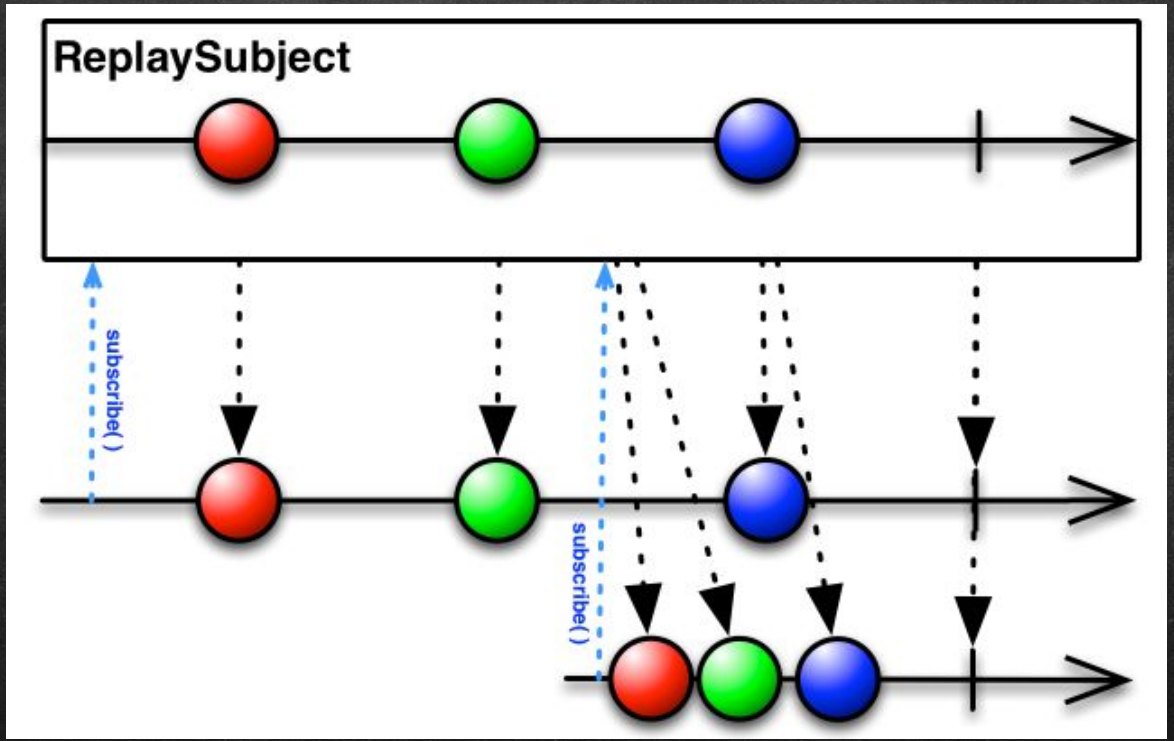
CRAFTSMEN

PublishSubject



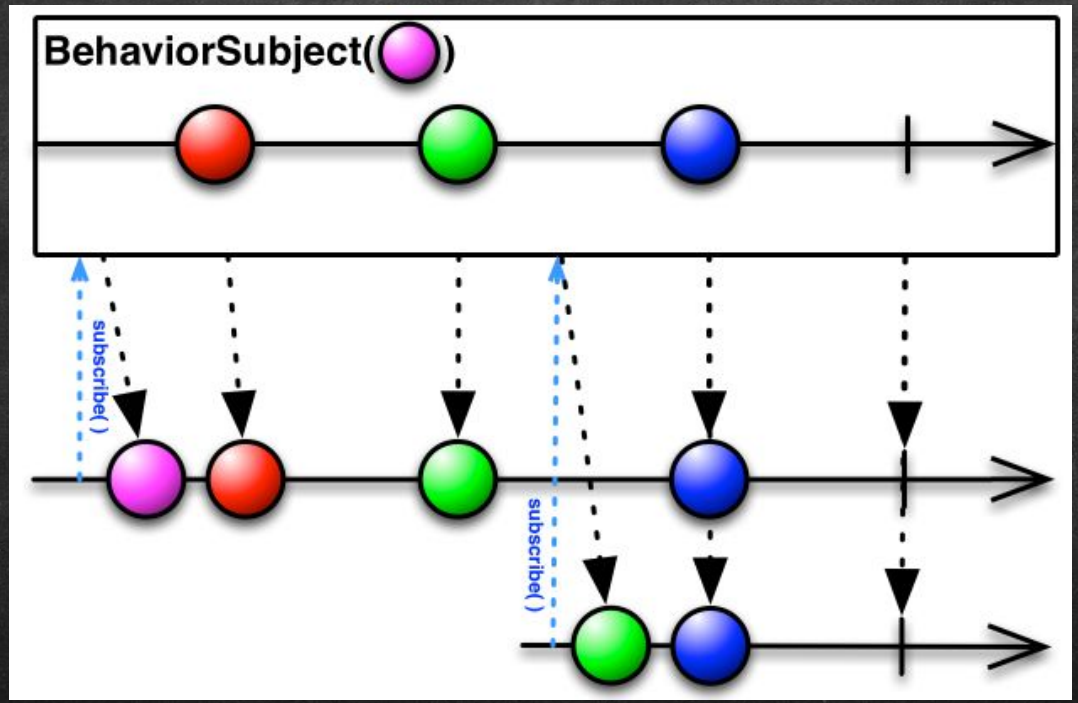
CRAFTSMEN

ReplaySubject

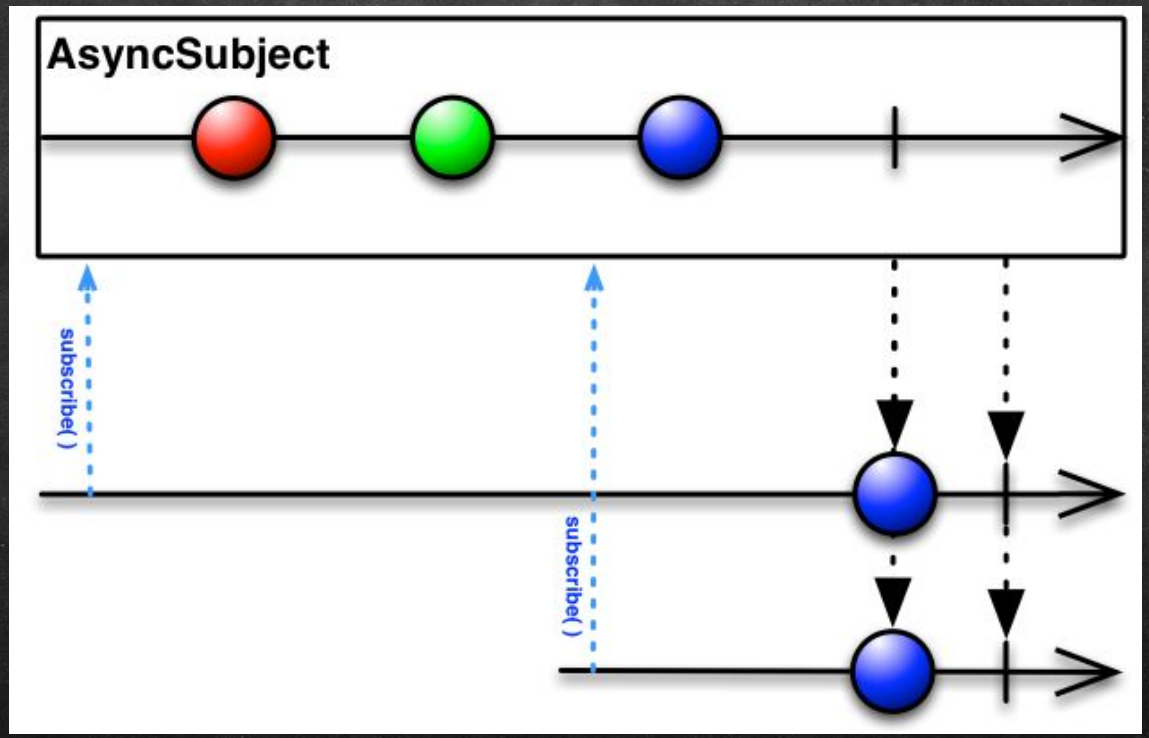


CRAFTSMEN

BehaviorSubject



AsyncSubject



CRAFTSMEN

Subject / ReplaySubject / BehaviorSubject

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

const subject = new Subject();

subject.subscribe((value) => console.log(`subscriber 1: ${value}`));
subject.next(1);
subject.next(2);
subject.subscribe((value) => console.log(`subscriber 2: ${value}`));
subject.next(3);
subject.complete();
subject.subscribe((value) => console.log(`subscriber 3: ${value}`));
```

CRAFTSMEN

Hot vs Cold Observables

- Hot: emits values regardless of subscription
- Cold: starts emitting after subscription from observer
- Transform cold observable into hot observable with publish operator

CRAFTSMEN

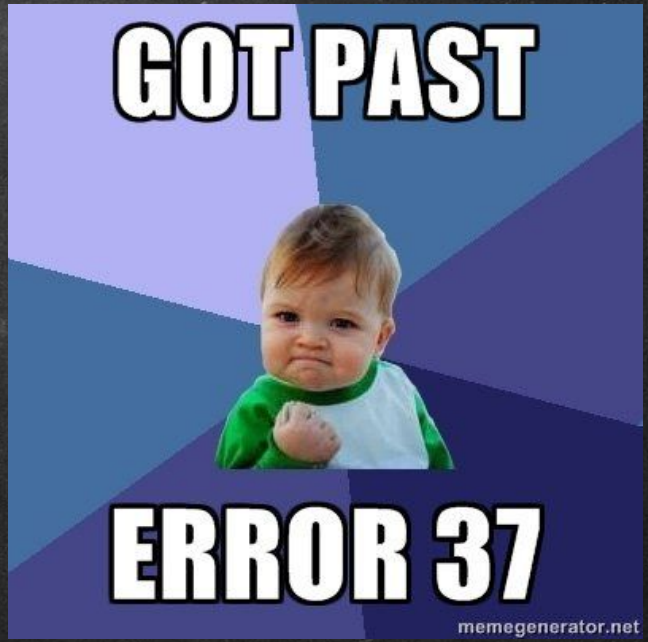
Hot vs Cold Observables

```
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';

const values$ = interval(500).pipe(take(10));

values$.subscribe((value) => console.log(`a: ${value}`));

setTimeout(
    () => values$.subscribe((value) => console.log(`b: ${value}`)),
    3000
);
```

CRAFTSMEN

Error handling

- Errors get emitted in a different channel
- Stream terminates when an error occurs
- Various recovery possibilities
  - Recover by switching to a different observable
  - Restart immediately / after some delay
  - Let subscriber deal with error



CRAFTSMEN

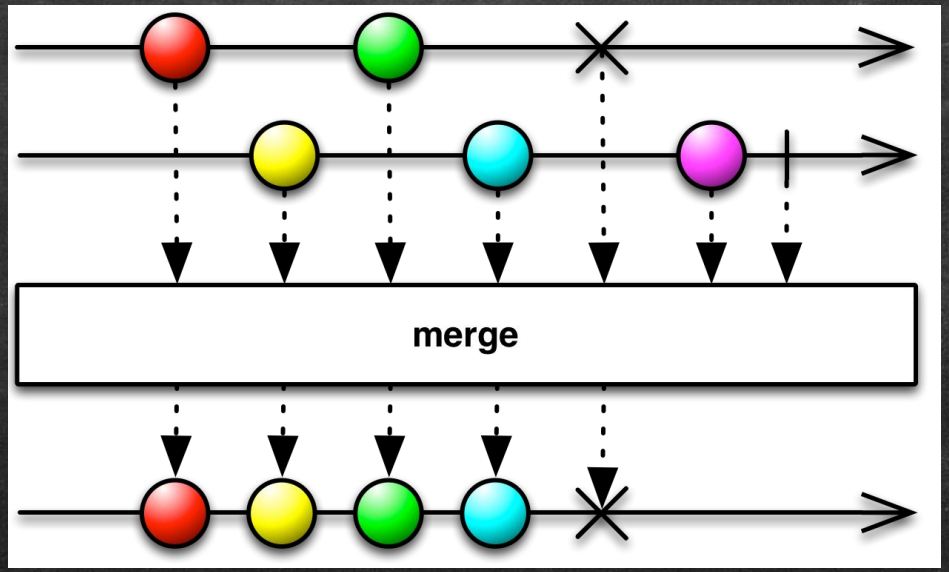
Error handling - fallback with `catchError`

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

of('one', 'two', null, 'four')
    .pipe(
        map((value) => value.toUpperCase()),
        catchError((error) => of('TEN', 'ELEVEN'))
    )
    .subscribe(console.log, console.error);
```

CRAFTSMEN

Operators part II

merge



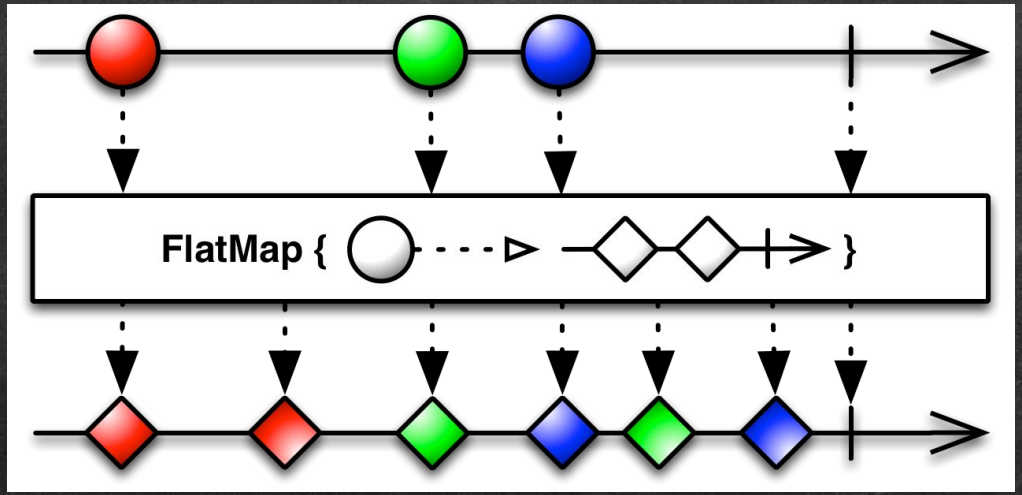
CRAFTSMEN

merge

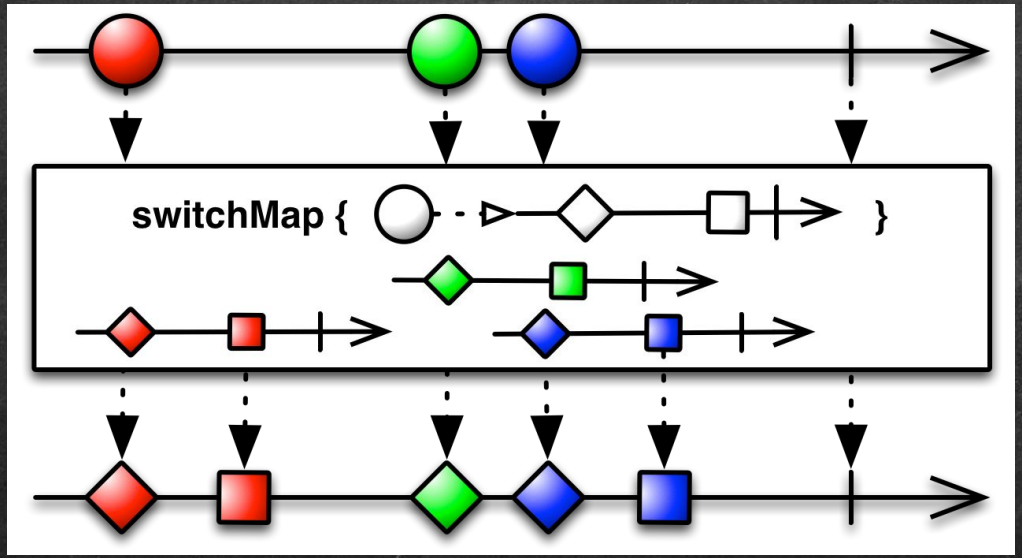
```
import { of, merge } from 'rxjs';
import { delay } from 'rxjs/operators';

merge(
    of('a').pipe(delay(100)),
    of('b').pipe(delay(200)),
    of('c').pipe(delay(500)),
    of('d').pipe(delay(800)),
    of('e').pipe(delay(850))
).subscribe(console.log);
```

`mergeMap` (a.k.a. `flatMap`)

`mergeMap` (a.k.a. `flatMap`)



switchMap

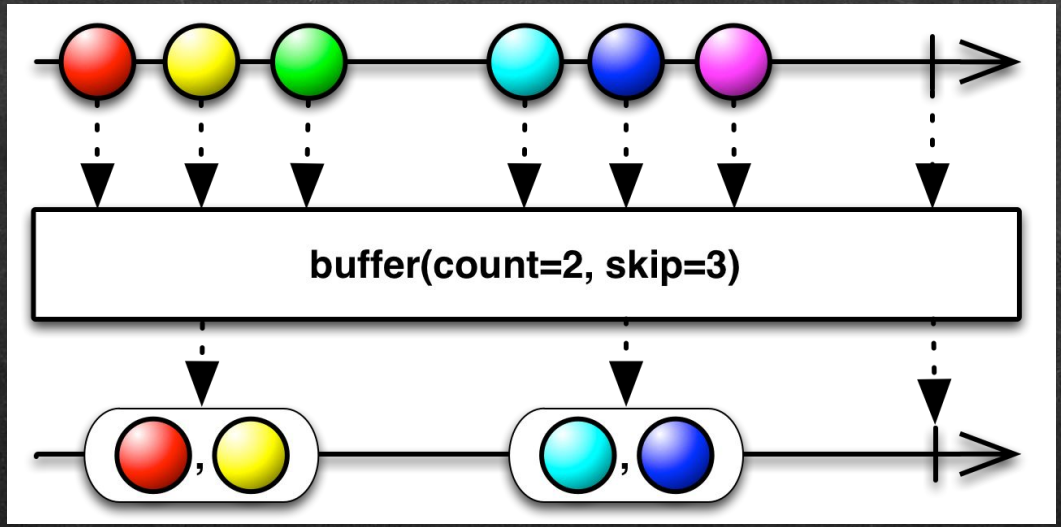


CRAFTSMEN

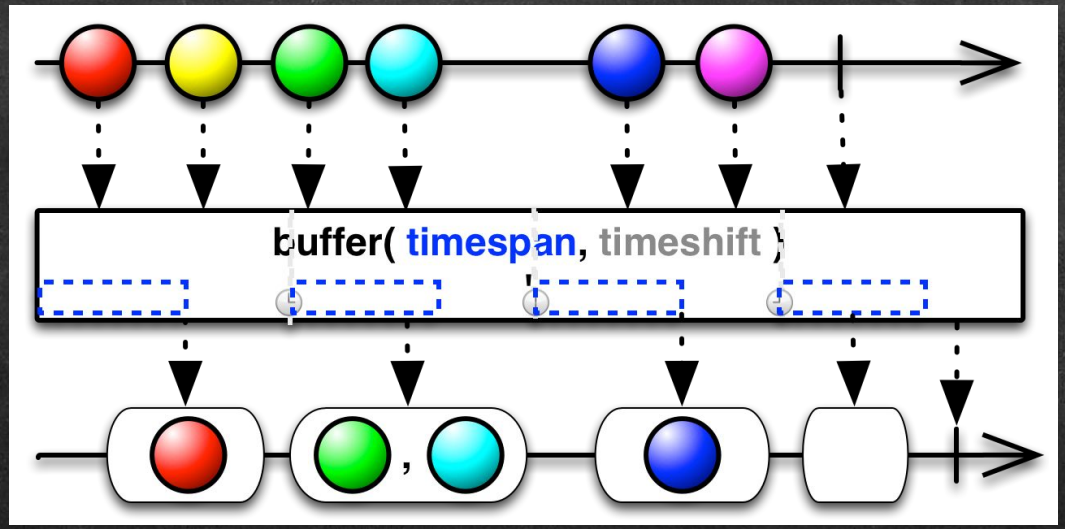
mergeMap

```
import { interval } from 'rxjs';
import { take, map, mergeMap } from 'rxjs/operators';

const number$ = interval(1000).pipe(take(5), map((value) => value + 1));
const letter$ = interval(3000).pipe(take(3), map((value) =>
    String.fromCharCode(value + 65)));

letter$
    .pipe(
        mergeMap((letter) => number$.pipe(
            map((number) => `${letter}-${number}`)
        ))
    )
    .subscribe(console.log);
```

bufferCount



bufferTime

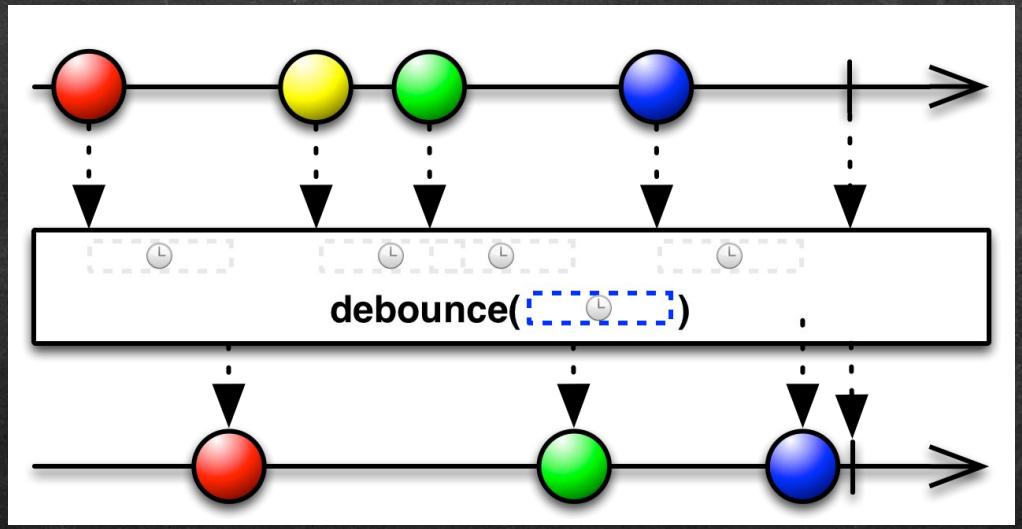


CRAFTSMEN

bufferCount

```
import { of } from 'rxjs';
import { bufferCount } from 'rxjs/operators';

const number$ = of('a', 'b', 'c', 'd', 'e');

number$
    .pipe(bufferCount(3, 2))
    .subscribe(console.log);
```

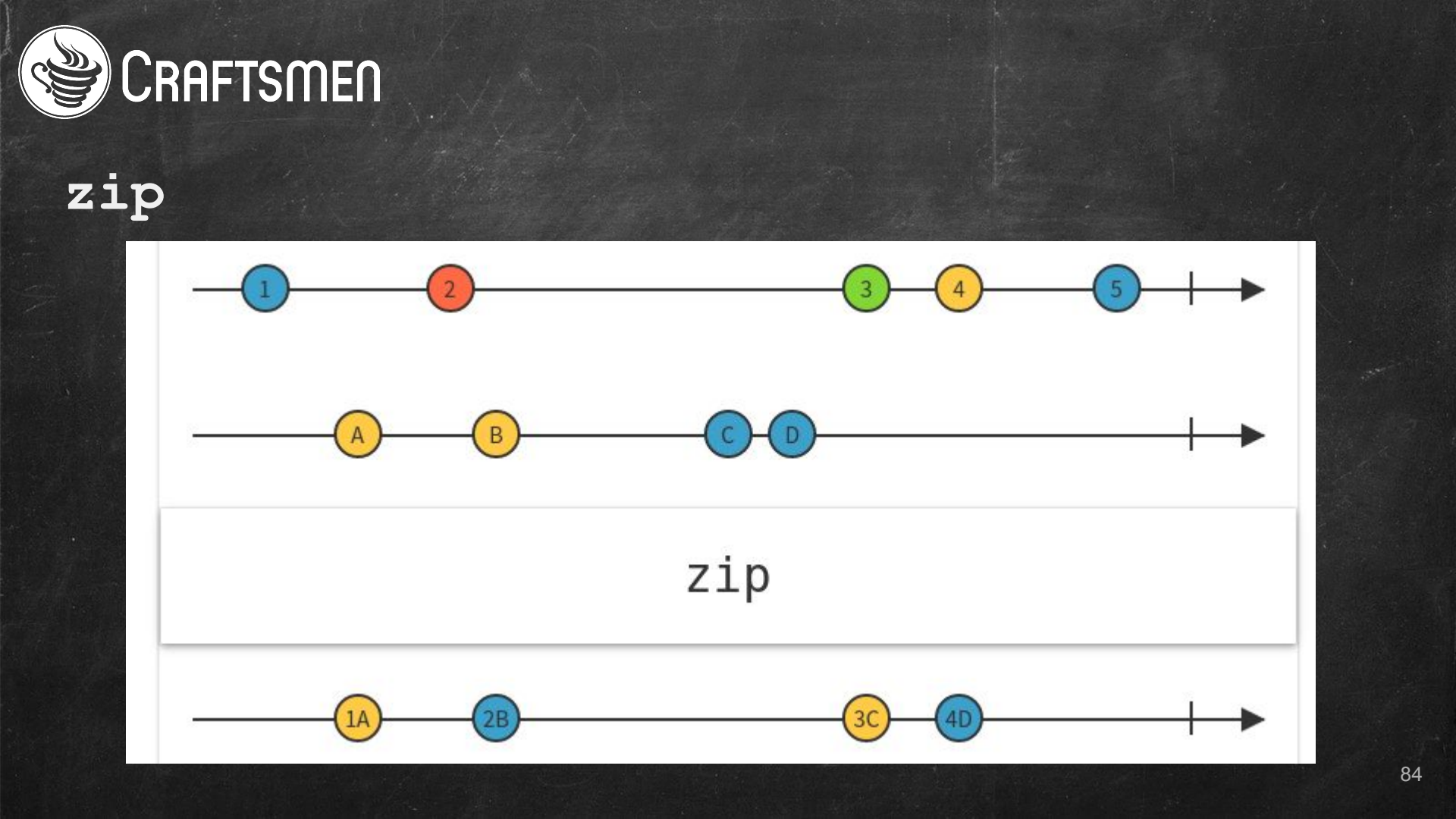
debounceTime



CRAFTSMEN

debounceTime

```
import { of, merge } from 'rxjs';
import { debounceTime, delay } from 'rxjs/operators';

const letter$ = merge(
   of('a').pipe(delay(100)),
   of('b').pipe(delay(200)),
   of('c').pipe(delay(500)),
   of('d').pipe(delay(800)),
   of('e').pipe(delay(850))
);

letter$
   .pipe(debounceTime(250))
   .subscribe(console.log);
```

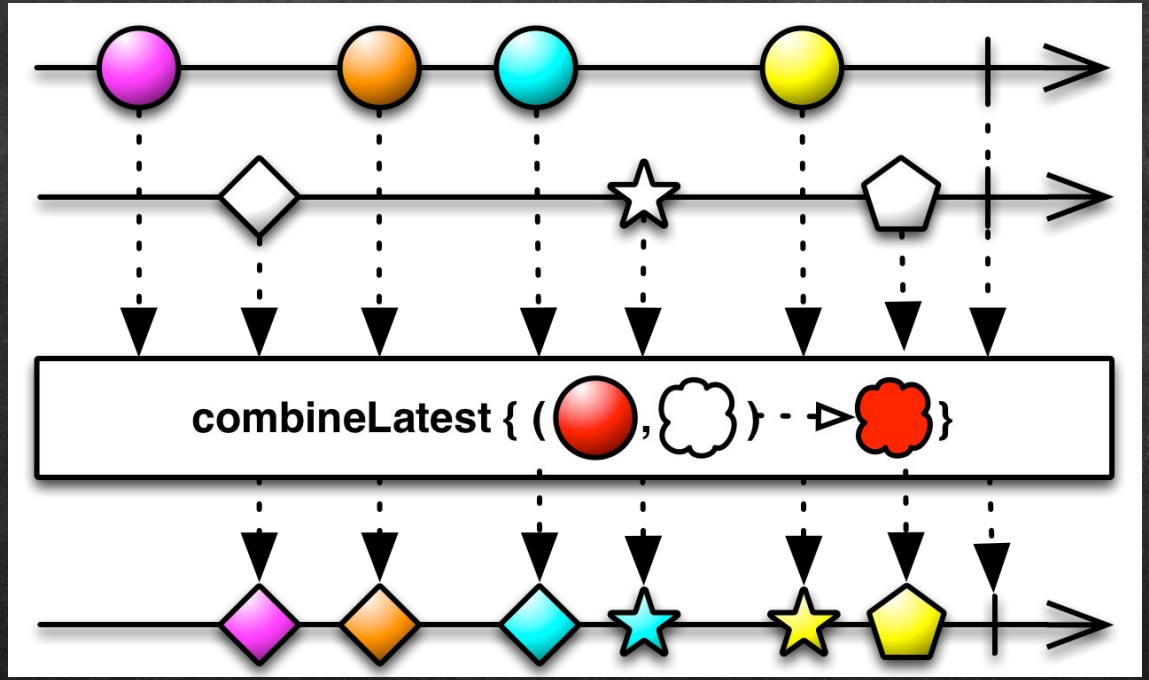

CRAFTSMEN
zip
1
2
3
4
5
A
B
C
D
zip
1A
2B
3C
4D

CRAFTSMEN

zip

```
import { of, zip } from 'rxjs';

const cash$ = of(50.0, 51.1, 48.3, 49.5);
const rate$ = of(1.06, 1.03, 1.04, 1.09);

zip(cash$, rate$)
    .subscribe(([cash, rate]) =>
        console.log(`Money: ${(cash * rate).toFixed(2)}`)
    );
```

combineLatest

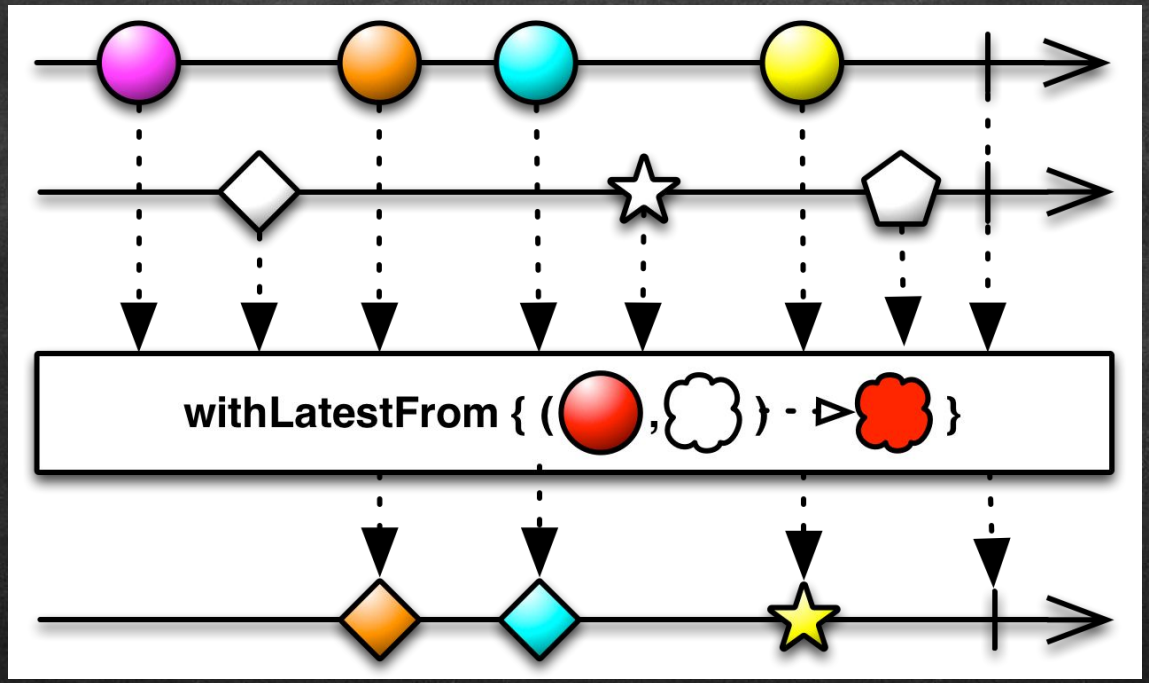


CRAFTSMEN

combineLatest

```
import { of, merge, combineLatest } from 'rxjs';
import { delay } from 'rxjs/operators';

const letter$ = merge(
    of('A').pipe(delay(100)),
    of('B').pipe(delay(300)),
    of('C').pipe(delay(400))
);

const number$ = merge(
    of(1).pipe(delay(200)),
    of(2).pipe(delay(500)),
)

combineLatest(letter$, number$).subscribe(console.log);
```

withLatestFrom



CRAFTSMEN

withLatestFrom

```
import { of, merge } from 'rxjs';
import { delay, withLatestFrom } from 'rxjs/operators';

const letter$ = merge(
   of('A').pipe(delay(100)),
   of('B').pipe(delay(300)),
   of('C').pipe(delay(400))
);

const number$ = merge(
   of(1).pipe(delay(200)),
   of(2).pipe(delay(500)),
)

letter$.pipe(withLatestFrom(number$)).subscribe(console.log);
```

CRAFTSMEN

Propagation of change - revisited

```
let x$ = 3;

let y$ = 4;

let z$ = x$ * y$;

console.log(z$); // 12

y$ = 3;

console.log(z$); // 9
```

```
let x$ = new BehaviorSubject(3);

let y$ = new BehaviorSubject(4);

let z$ = combineLatest(x$, y$)
  .pipe(map(([x, y]) => x * y));

z$.subscribe(console.log);

y$.next(3);
```

CRAFTSMEN

Workshop part II

- Time to start coding again
- Apply what you've learned from
  - Creating observables
  - "Advanced" operators
- To the domain of trains:
  - Check in / check out
  - Computing train speeds
  - Departures and arrivals
- Do workshop exercises **12** to **18**

CRAFTSMEN

Recap

- Reactive programming is focused on data flows and propagation of change
- ReactiveX is a polyglot RP library
- The observable is at the heart of ReactiveX
- An (a)synchronous data stream
  - Push based
  - Events: next (data), error and complete
  - Separated from their source
  - Composable (through operators)
- Well suited for I/O operations (due to their latency):
  - Database queries/updates, disk access, networking, user input devices

CRAFTSMEN

Stuff we haven’t covered

- Many more operators
  - Different flavours of presented operators
- Backpressure
- Schedulers
- Testing and debugging
- Check README.md for more material

CRAFTSMEN

Common pitfalls

- Reassigning observables
  - Observables should be defined only once and never* be reassigned (use `const` or `final`).
    ** as always there are some legitimate exceptions to this rule*
- Modifying application state within operators
  - This should only be done by a subscriber
  - So only use ***pure functions*** for composing observables!
- Assuming an observable is hot instead of cold
- Not unsubscribing from observables
- Usage of a mutable seed value for `scan` / `reduce` operators

CRAFTSMEN

Closing remarks

Domains in which observable streams are useful:

- Processing of live sensor data
- Networking applications (HTTP, streaming data)
- User interfaces
- I/O in general, due latency and its asynchronous nature

Tips for solving problems using Observables:

- Define your 'input' and desired 'output' observables, work your way from there
- Draw marble diagrams
- Familiarize yourself with the different operators

CRAFTSMEN