Reactive Programming

16 december 2016,
Daan Scheerens & Remco Weekers

CRAFTSMEN

About us

Remco Weekers

remco@craftsmen.nl

https://github.com/rweekers

https://www.strava.com/athletes/789756

Maastricht University -Knowledge engineering

Daan Scheerens

daan@craftsmen.nl

https://github.com/dscheerens/

University of Twente - Computer Science

Functional Programming and Reactive Programming enthusiast

@craftsmen_nl

Program

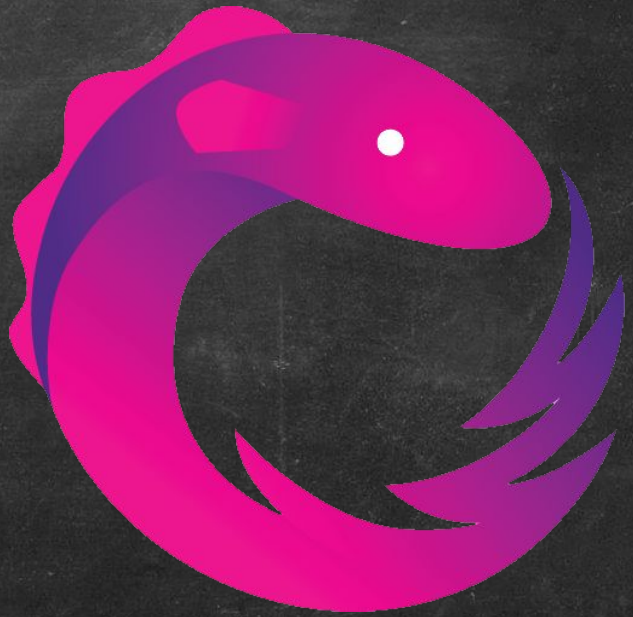
14:30 - Introduction in RP / ReactiveX

15:00 - Workshop part I

15:50 - Diving deeper into ReactiveX

16:30 - Workshop part II

17:15 - Wrap up and drinks!



Why Reactive Programming?

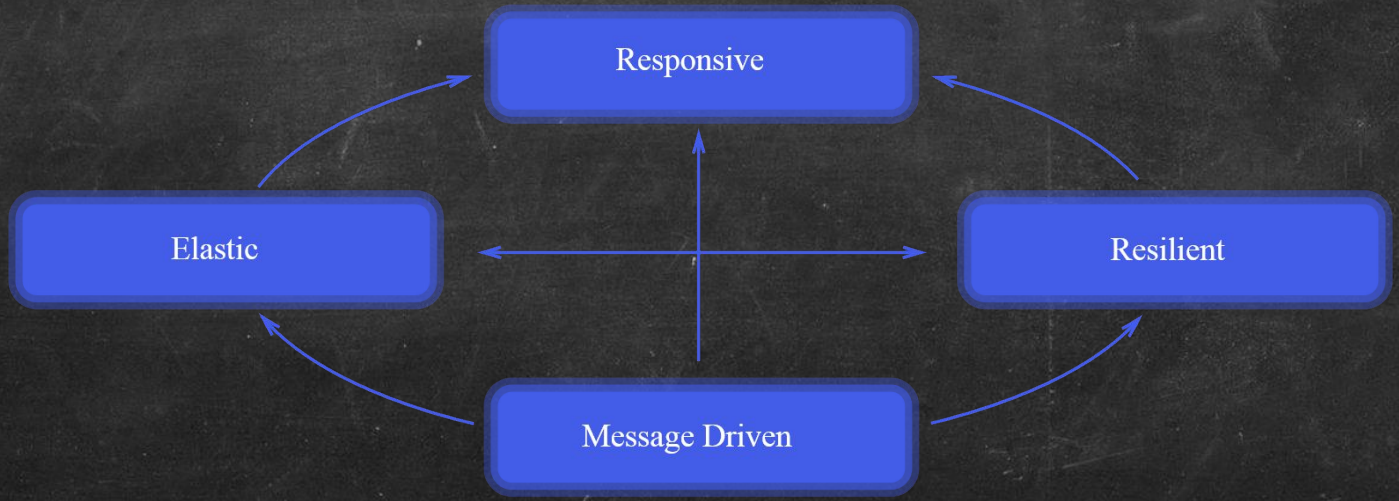
- It is a hot topic
- Increased adoption in frameworks and libraries
- Will make your life (and that of your colleagues) easier

Reactive Systems

vs

Reactive Programming

The reactive manifesto



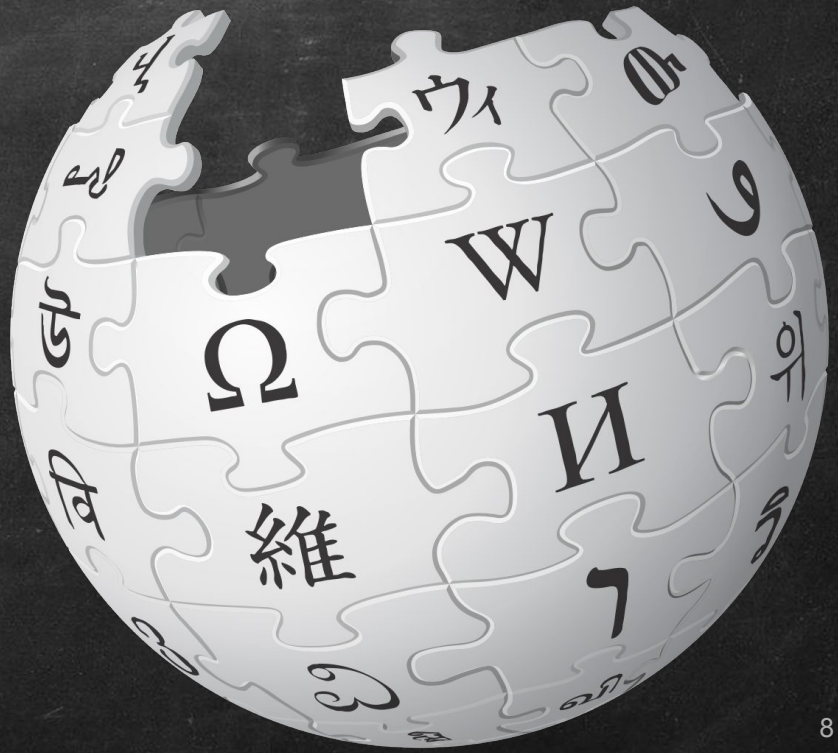
Reactive Systems

- Focus lies on application architecture that enables:
  - High availability
  - Scalability
  - Responsiveness
- Typical frameworks/libraries
  - Vert.x
  - Akka
  - Apache Spark
  - *Spring 5?*
  - *… many more*

CRAFTSMEN

Wikipedia

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



Reactive Programming

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

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

Propagation of change

Reactive Programming libraries

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

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

ReactiveX family overview

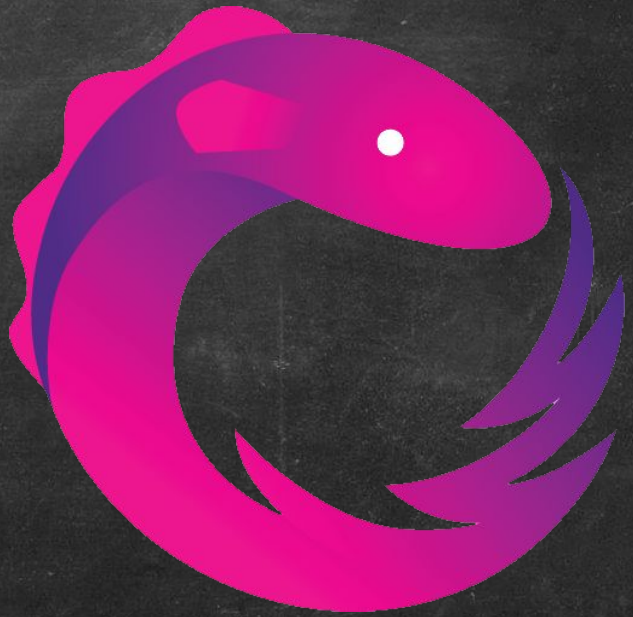
Java:	**RxJava**	Ruby:	**Rx.rb**
JavaScript:	**RxJS**	Python:	**RxPY**
C#:	**Rx.NET**	Groovy:	**RxGroovy**
C#(Unity):	**UniRx**	JRuby:	**RxJRuby**
Scala:	**RxScala**	Kotlin:	**RxKotlin**
Clojure:	**RxClojure**	Swift:	**RxSwift**
C++:	**RxCpp**	PHP:	**RxPHP**
Lua:	**RxLua**		

RxJS 5 got released this week!

ReactiveX

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

http://reactivex.io/



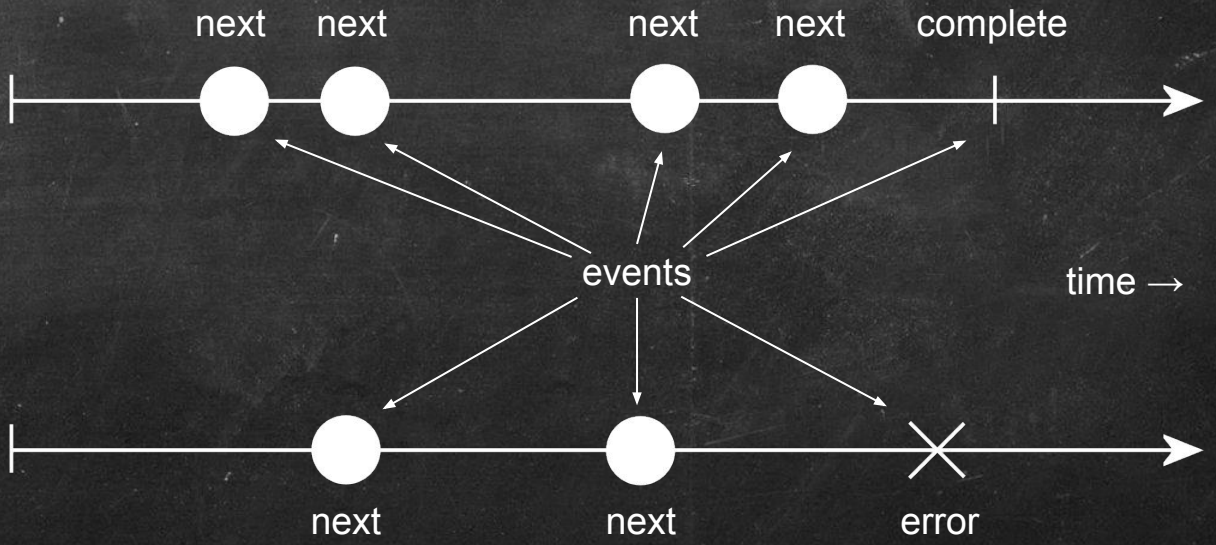
The observable:

Abstract stream of (a)synchronous data

Events:

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

Subscribe to observable
to receive events!



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

Warning

Java 8 / ES6 Lambda functions ahead!

CRAFTSMEN

Subscribing to observables

```java
Observable<String> message$ = giveMeData();

message$.subscribe(
  (value) -> System.out.println("next: " + value),
  (error) -> System.out.println("error: " + error),
  ()      -> System.out.println("complete")
);
```

Java

```javascript
const message$ = giveMeData();

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

JavaScript

Observer objects - JavaScript

Alternative method to subscribe to Observable events

```
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

Observer<T> interface (Java)

Alternative method to subscribe to Observable events

```
Observable<String> tweets$ = fetchTweets();

Observer<String> observer = new Observer<String>() {
  @Override public void onNext(String value) {
    System.out.println("next: " + value);
  }
  @Override public void onError(Throwable error) {
    System.out.println("error: " + error);
  }
  @Override public void onCompleted() {
    System.out.println("complete");
  }
};

tweets$.subscribe(observer);
```

Relation with Observable pattern

Similarities

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

  ** unsubscription may happen automatically for observable streams*

Differences

- Observable streams can be passed around as *first-class citizens*
- Observable streams can be composed
- Observable streams have separate error and complete channels

Relation with Iterator pattern

Similarities

- Both can be passed around as *first-class citizens*
- 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

Relation with Java 8 streams

Similarities

- Both can be passed around as *first-class citizens*
- Both can be composed

Differences

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

Relation with JavaScript (ES6) Promise

Similarities

- Both can be passed around as *first-class citizens*
- Both can be asynchronous
- Both have a separate error channel

Differences

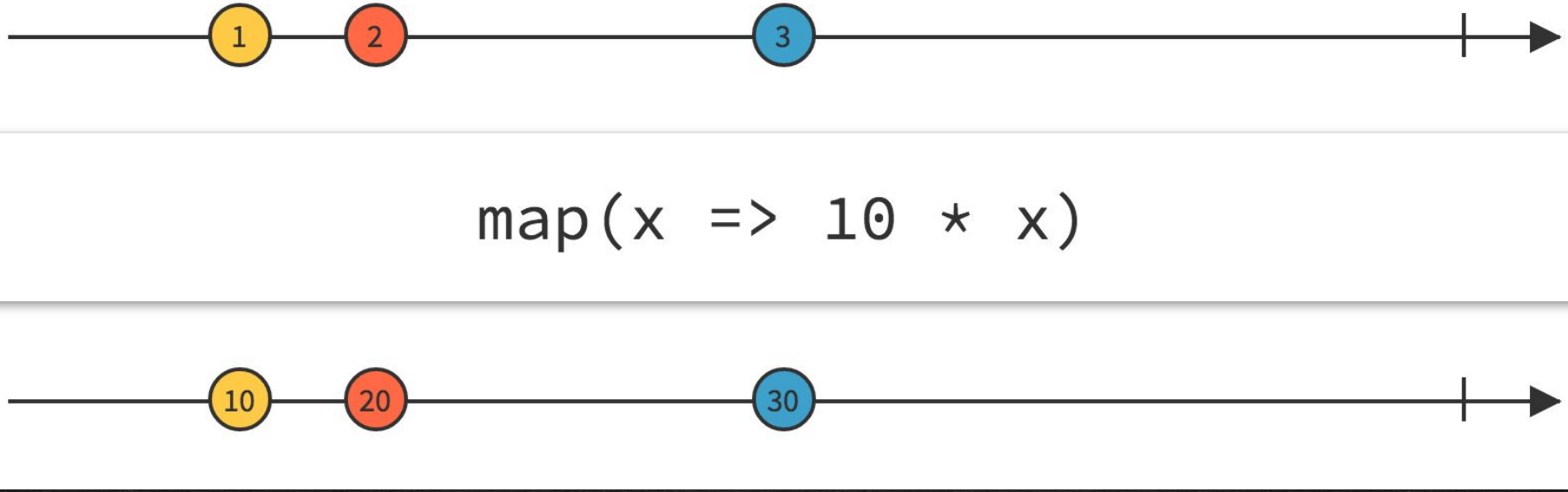
- Observable streams support multiple values (over time)
- Observable streams don’t store history*, Promises do

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

CRAFTSMEN

Composing observable streams: operators

Observable.map()



Observable.map()

```java
Integer[] a = { 1, 2, 3, 4 };
Observable<Integer> o = Observable.from(a);

o.map((x) -> x * 10).subscribe(System.out::println);
```

Java

```javascript
const o = Rx.Observable.of(1, 2, 3, 4);

o.map((x) => x * 10).subscribe(console.log);
```

JavaScript

CRAFTSMEN

Observable.filter()


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

Observable.filter()

```java
Integer[] a = { 1, 2, 3, 4 };
Observable<Integer> o = Observable.from(a);

o.filter((x) -> x % 2 == 0).subscribe(System.out::println);
```

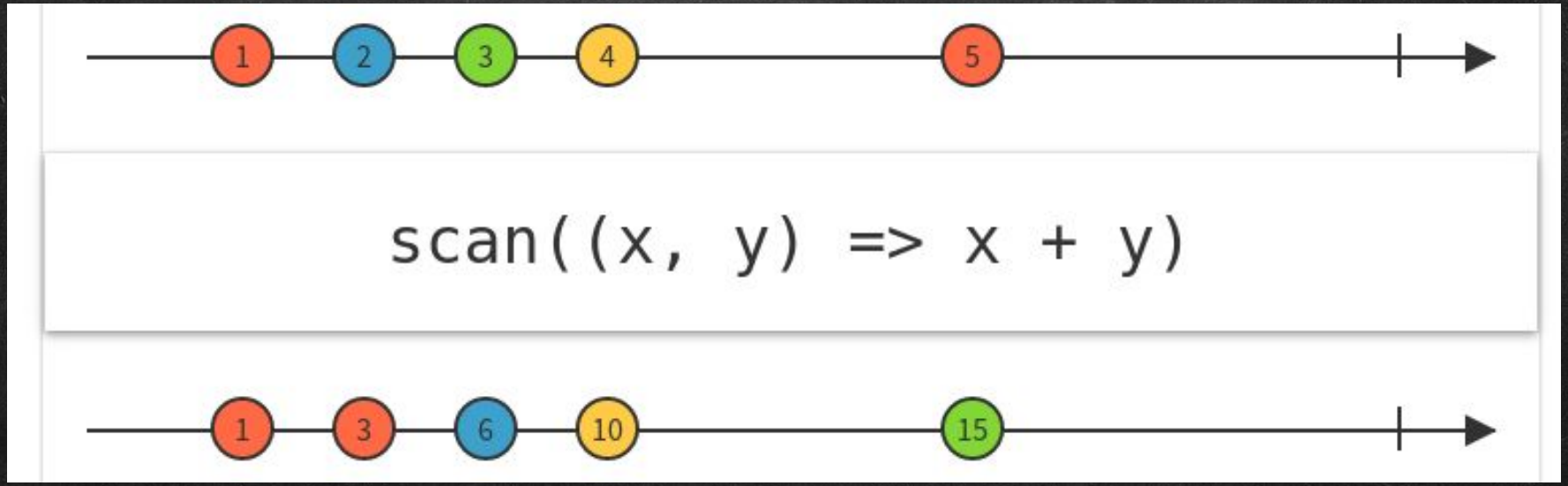
Java

```javascript
const o = Rx.Observable.of(1, 2, 3, 4);

o.filter((x) => x % 2 == 0).subscribe(console.log);
```

JavaScript

CRAFTSMEN

Observable.scan()



Observable.scan()

```java
Integer[] a = { 1, 2, 3, 4 };
Observable<Integer> o = Observable.from(a);

o.scan((x, y) -> x + y).subscribe(System.out::println);
```

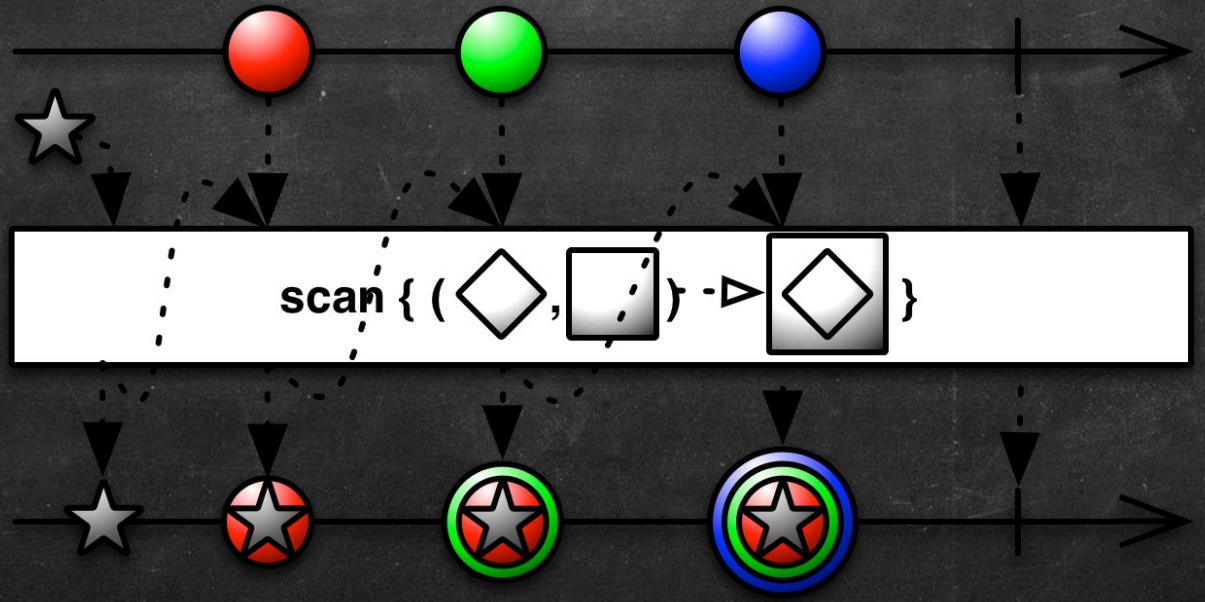
Java

```javascript
const o = Rx.Observable.of(1, 2, 3, 4);

o.scan((x, y) => x + y).subscribe(console.log);
```

JavaScript

CRAFTSMEN

`Observable.scan()` *with seed value*



`Observable.scan()` *with seed value*

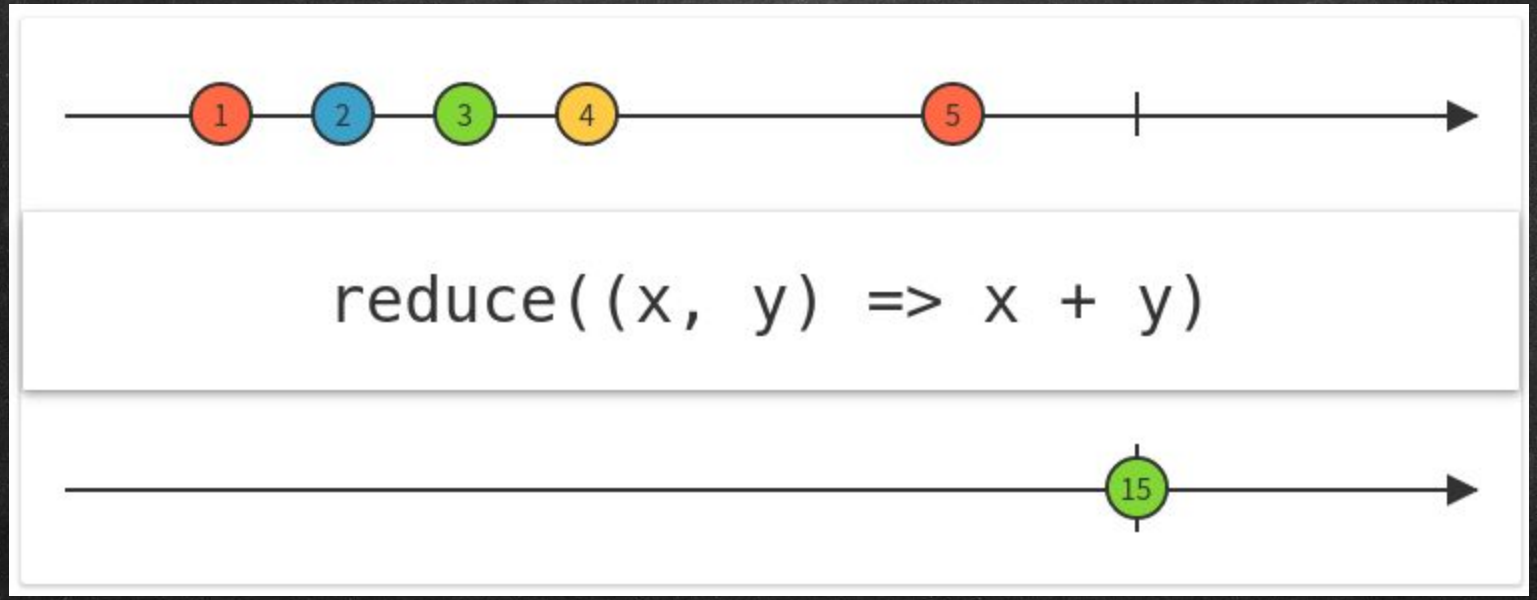
```java
Observable.from(new Integer[] { 1, 2, 3, 4 })
  .scan(100, (x, y) -> x + y)
  .subscribe(System.out::println);
```

Java

```javascript
Rx.Observable.of(1, 2, 3, 4)
     .scan((x, y) => x + y, 100)
     .subscribe(console.log);
```

JavaScript

Observable.reduce()



Observable.reduce()

```java
Integer[] a = { 1, 2, 3, 4 };
Observable<Integer> o = Observable.from(a);

o.reduce((x, y) -> x + y).subscribe(System.out::println);
```

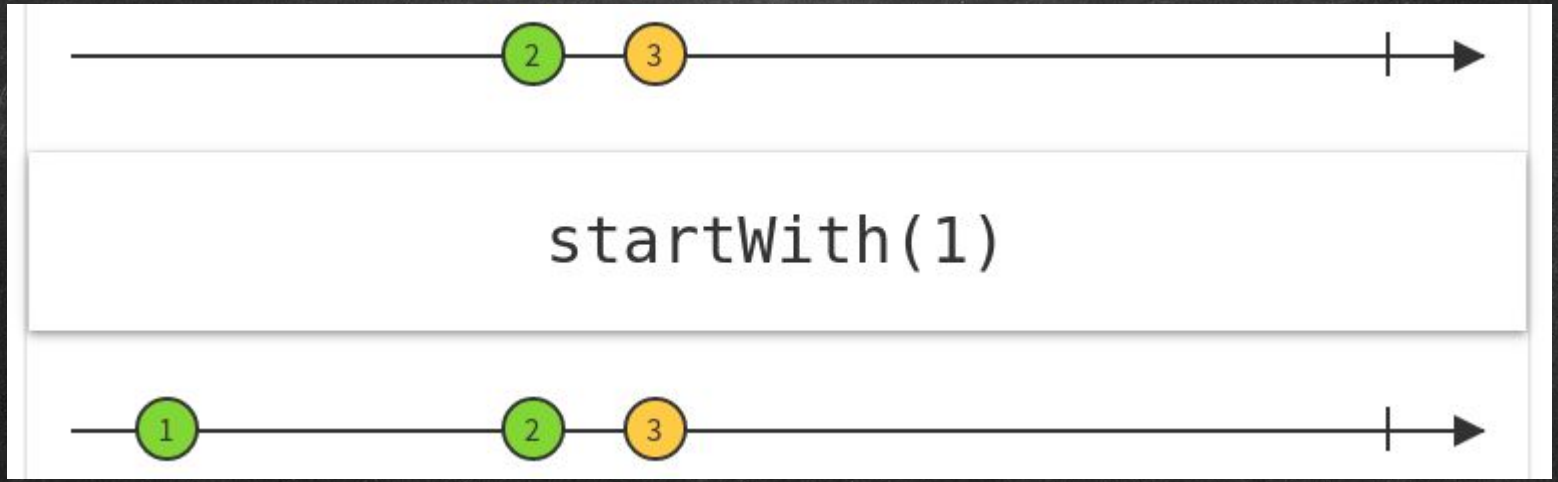
Java

```javascript
const o = Rx.Observable.of(1, 2, 3, 4);

o.reduce((x, y) => x + y).subscribe(console.log);
```

JavaScript

CRAFTSMEN

Observable.startWith()



Observable.startWith()

```java
Integer[] a = { 1, 2, 3, 4 };
Observable<Integer> o = Observable.from(a);

o.startWith(-2, -1, 0).subscribe(System.out::println);
```

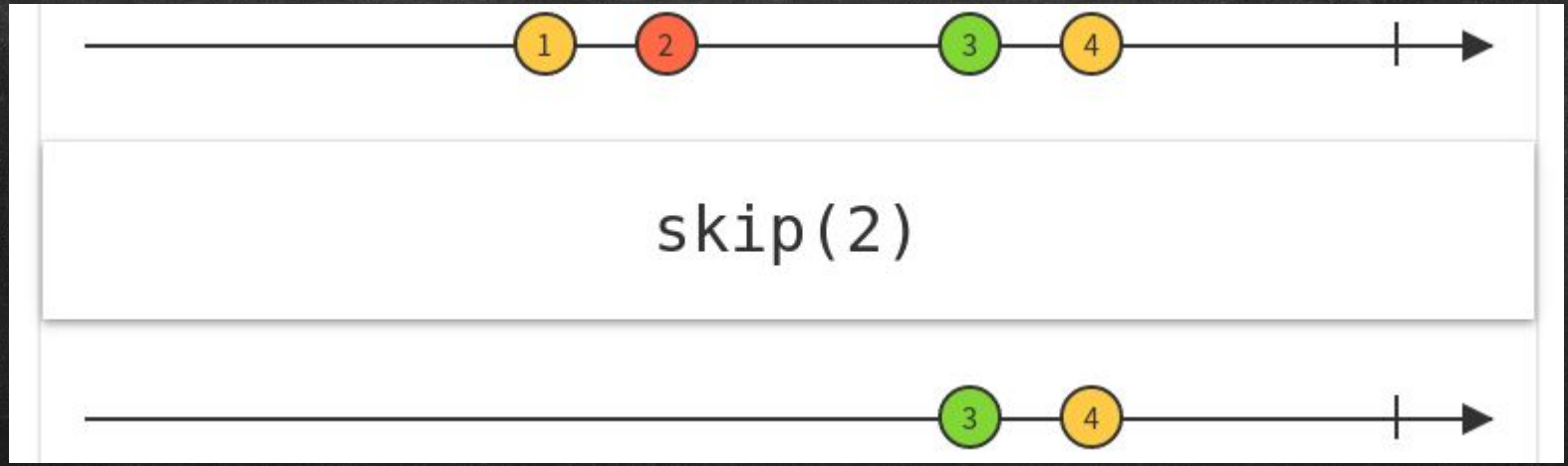
Java

```javascript
const o = Rx.Observable.of(1, 2, 3, 4);

o.startWith(-2, -1, 0).subscribe(console.log);
```

JavaScript

CRAFTSMEN

Observable.skip()



Observable.skip()

```java
Integer[] a = { 1, 2, 3, 4 };
Observable<Integer> o = Observable.from(a);

o.skip(2).subscribe(System.out::println);
```

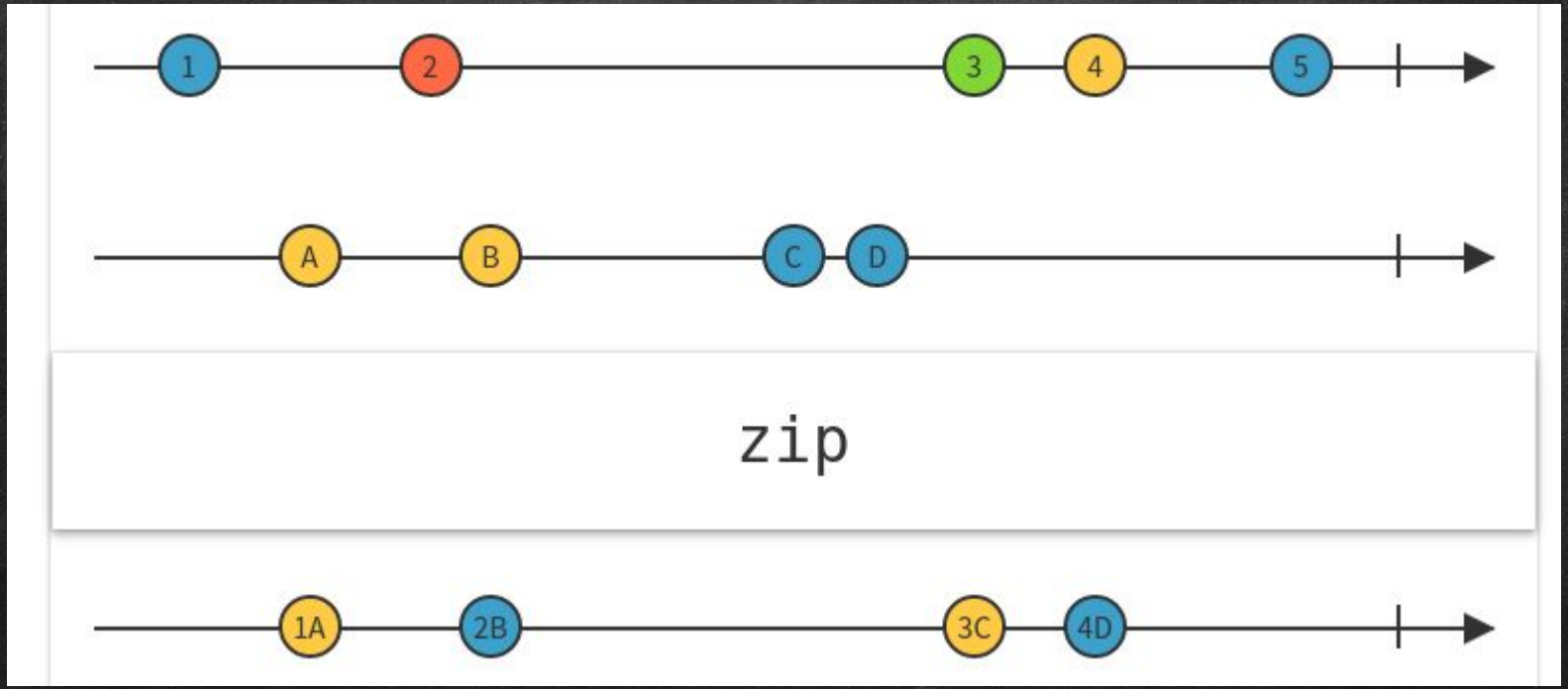
Java

```javascript
const o = Rx.Observable.of(1, 2, 3, 4);

o.skip(2).subscribe(console.log);
```

JavaScript

CRAFTSMEN

Observable.zip()



Observable.zip()

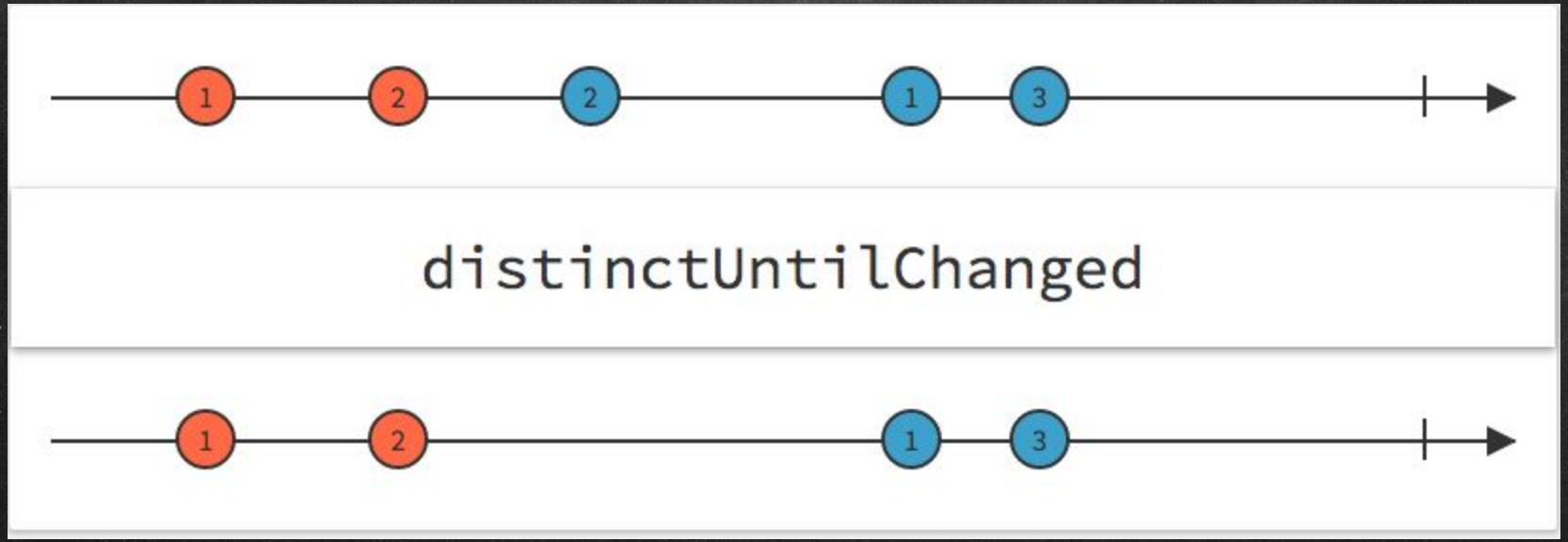
```java
Double[] cash = { 50.0, 51.1, 48.3, 49.5 };
Observable<Double> cash$ = Observable.from(cash);

Double[] rate = { 1.06, 1.03, 1.04, 1.09 };
Observable<Double> rate$ = Observable.from(rate);

cash$.zipWith(rate$, (change, stock) ->
    "Money to spend: " + String.format("%.2f", change * stock))
    .subscribe(System.out::println);
```

Java

```javascript
const cash$ = Rx.Observable.of(50.0, 51.1, 48.3, 49.5);
const rate$ = Rx.Observable.of(1.06, 1.03, 1.04, 1.09);

cash$.zip(rate$, (change, stock) =>
    'Money to spend: ' + (change * stock).toFixed(2))
    .subscribe(console.log);
```

JavaScript

Observable.distinctUntilChanged()



Observable.distinctUntilChanged()

```java
Integer[] a = { 1, 2, 2, 3, 3, 4};
Observable<Integer> o = Observable.from(a);

o.distinctUntilChanged().subscribe(System.out::println);
```

Java

```javascript
const o = Rx.Observable.of(1, 2, 2, 3, 3, 4);

o.distinctUntilChanged().subscribe(console.log);
```

JavaScript

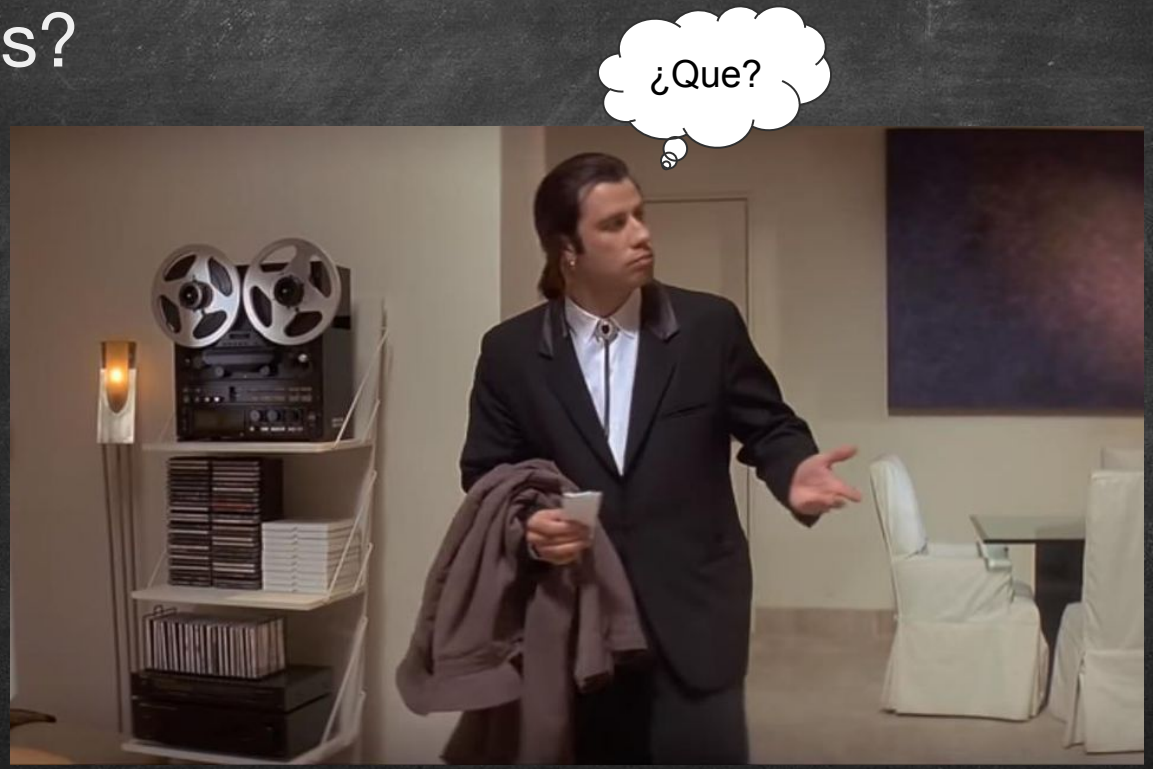
CRAFTSMEN

Live Demo!

Observations

- Composes very well
- Declarative style
- Easy to read / understand
- No callback hell / pyramid of doom
- No global state

Questions?



CRAFTSMEN

Time to start coding!

Workshop reactive operators:

- do math,
- linguistics
- ...and cooking in ReactiveX!
- find all info for `Java` and `JavaScript` on:
- https://gitlab.com/craftsmen/reactive-meetup
- assignments are found in code
- and operators are found on:
- http://reactivex.io/documentation/operators.html
- Part 2 of the presentation starts @ 15:50

Creating observables

Creation can be done in several ways:

- Static methods
- Using Subjects
- Custom coding

CRAFTSMEN

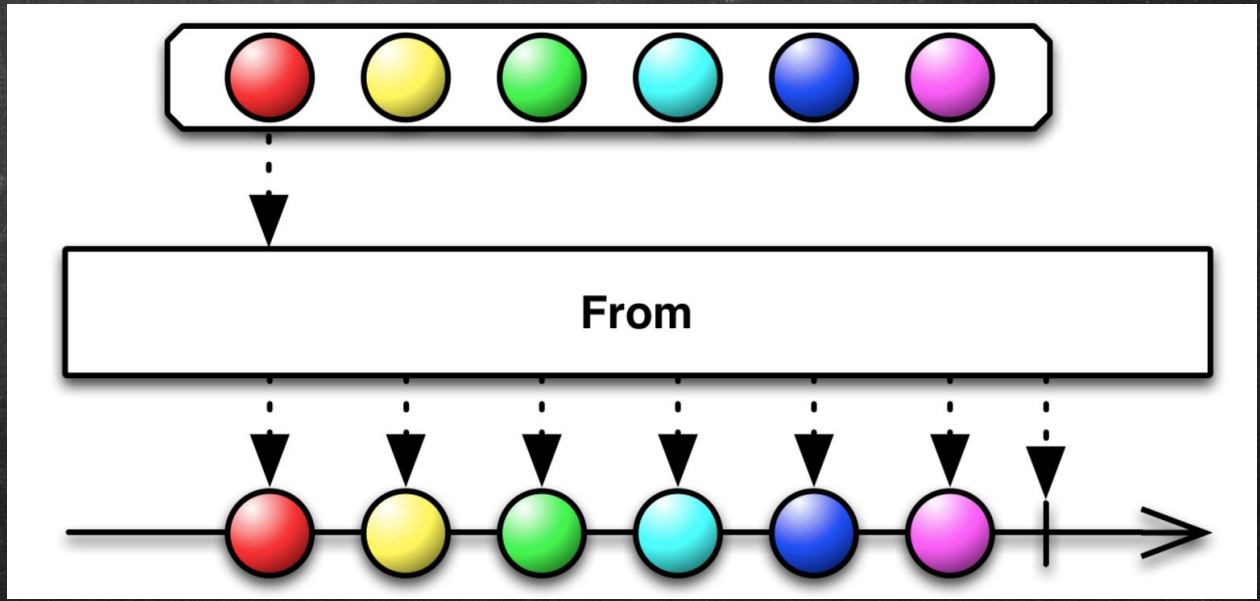
Creating observables - Static

Creation can be done with several methods:

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

Creating observables - From

Converts various objects into observables



Creating observables - From

```java
Integer[] a = { 1, 2, 3, 4 };
Observable<Integer> o = Observable.from(a);

o.subscribe(System.out::println);
```

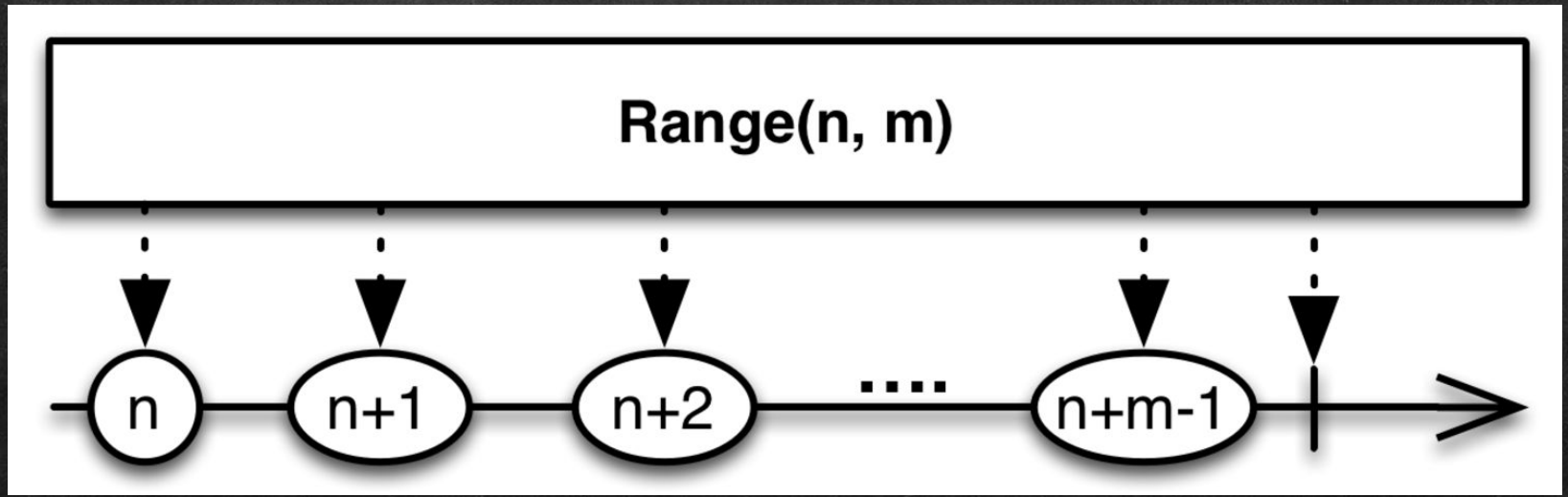
Java

```javascript
const o = Rx.Observable.of(1, 2, 3, 4);

o.subscribe(console.log);
```

JavaScript

Creating observables - Range

Emits a range of sequential integers



Creating observables - Range

```java
Observable<Integer> o = Observable.range(0, 10);
o.subscribe(System.out::println);
```

Java

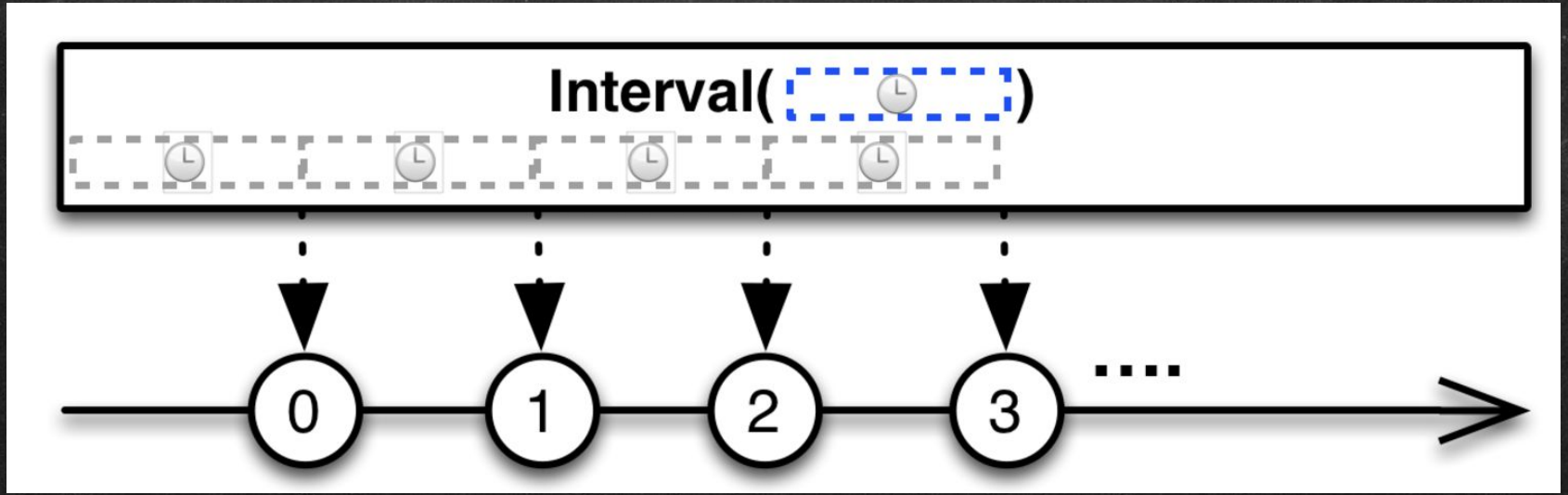
```javascript
const o = Rx.Observable.range(0, 10);
o.subscribe(console.log);
```

JavaScript

CRAFTSMEN

Creating observables - Interval

Emits a sequence of integers spaced by a time interval



Creating observables - Interval

```java
Observable<Long> o = Observable.interval(500, TimeUnit.MILLISECONDS);
o.subscribe(System.out::println);
```

Java

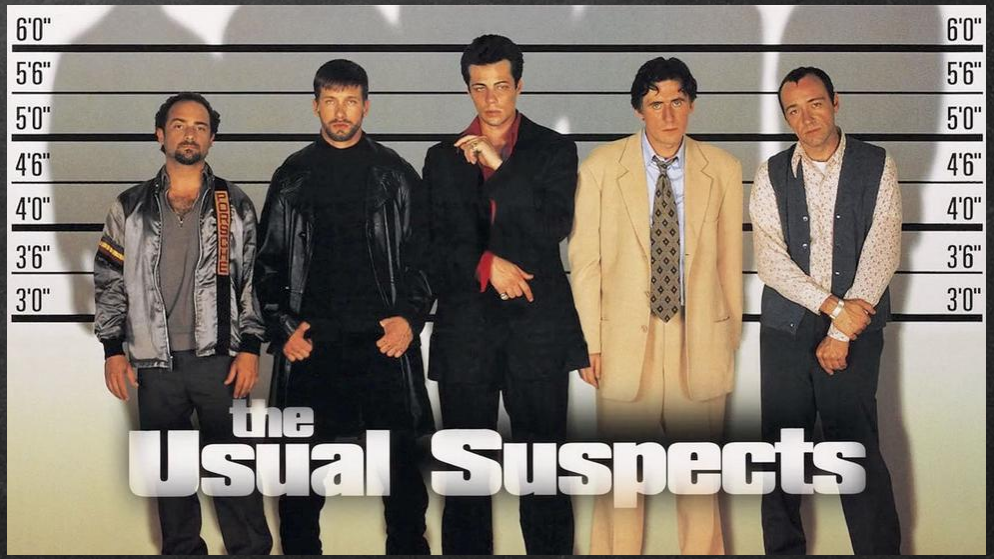
```javascript
const o = Rx.Observable.interval(500);
o.subscribe(console.log);
```

JavaScript

Creating observables - Subjects

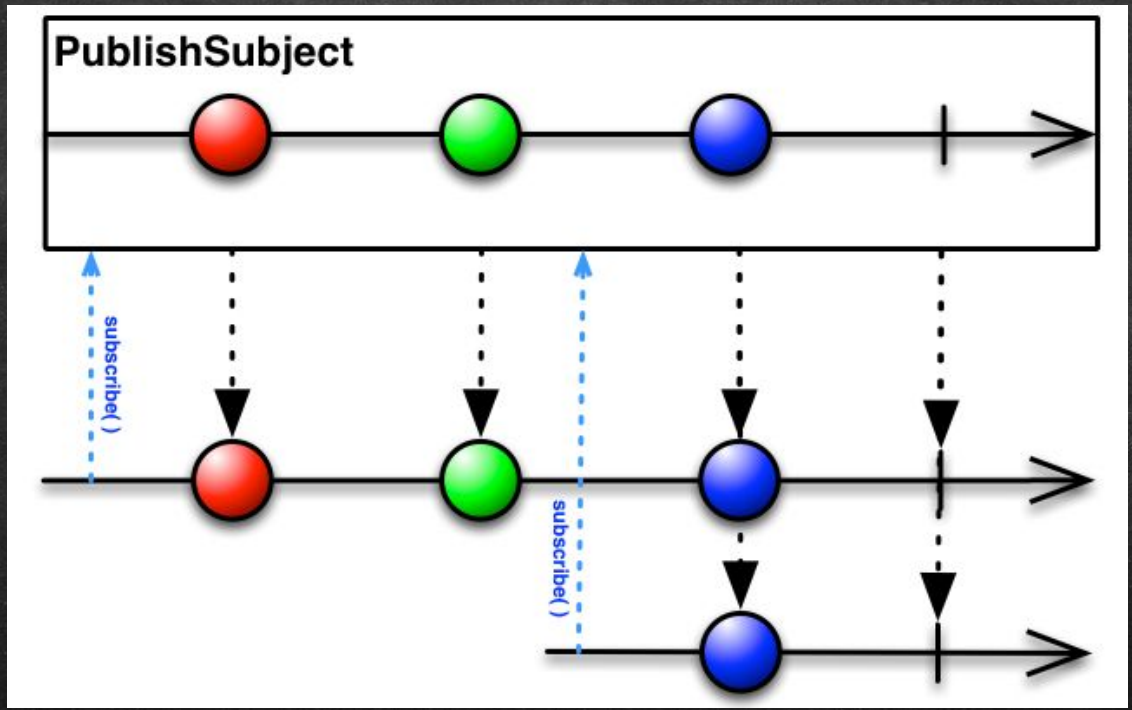
Several implementations available:

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



CRAFTSMEN

PublishSubject



PublishSubject

```java
PublishSubject<Integer> intSubject = PublishSubject.create();
Observable<Integer> intObserver = intSubject.asObservable();
intObserver.subscribe((v) -> System.out.println("Subscriber 1: " + v));
intSubject.onNext(1);
intSubject.onNext(2);

intObserver.subscribe((v) -> System.out.println("Subscriber 2: " + v));
intSubject.onNext(3);
intSubject.onCompleted();
```

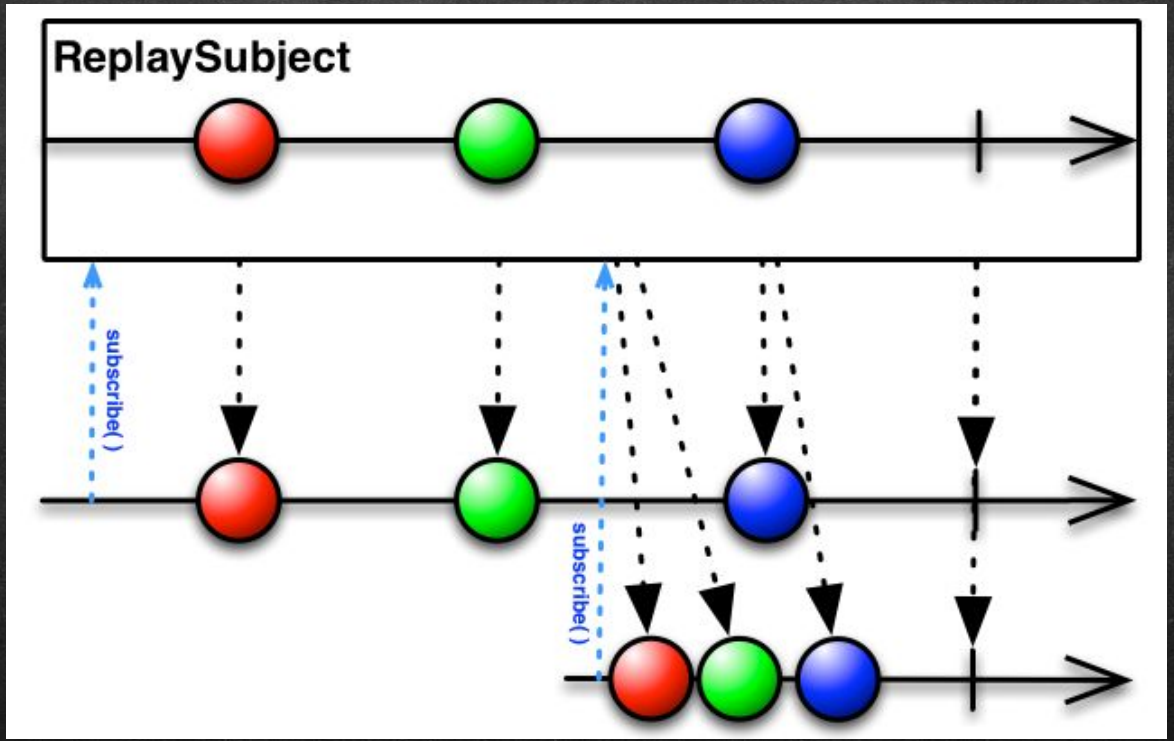
Java

```javascript
const subject = new Rx.Subject();
const observer = subject.asObservable();

observer.subscribe((v) => console.log('Subscriber 1: ' + v));
subject.next(1);
subject.next(2);

observer.subscribe((v) => console.log('Subscriber 2: ' + v));
subject.next(3);
subject.complete();
```

JavaScript

CRAFTSMEN

ReplaySubject



ReplaySubject

```java
ReplaySubject<Integer> intSubject = ReplaySubject.create();
Observable<Integer> intObserver = intSubject.asObservable();
intObserver.subscribe((v) -> System.out.println("Subscriber 1: " + v));
intSubject.onNext(1);
intSubject.onNext(2);

intObserver.subscribe((v) -> System.out.println("Subscriber 2: " + v));
intSubject.onNext(3);
intSubject.onCompleted();
```

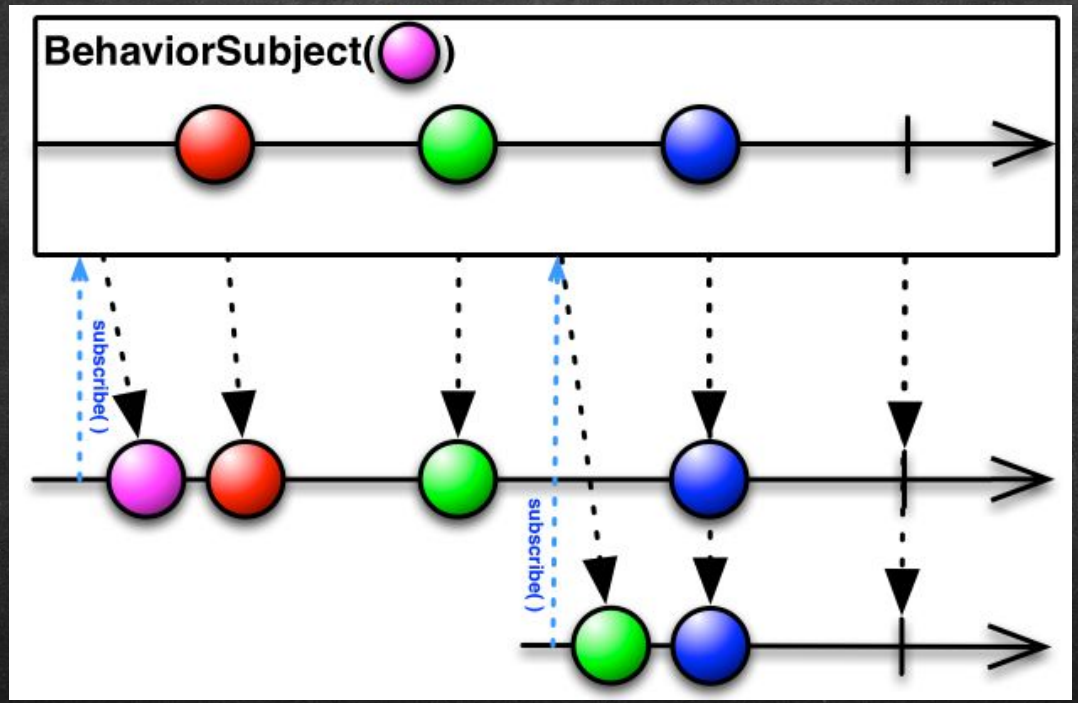
Java

```javascript
const subject = new Rx.ReplaySubject();
const observer = subject.asObservable();

observer.subscribe((v) => console.log('Subscriber 1: ' + v));
subject.next(1);
subject.next(2);

observer.subscribe((v) => console.log('Subscriber 2: ' + v));
subject.next(3);
subject.complete();
```

JavaScript

CRAFTSMEN

BehaviorSubject



BehaviorSubject

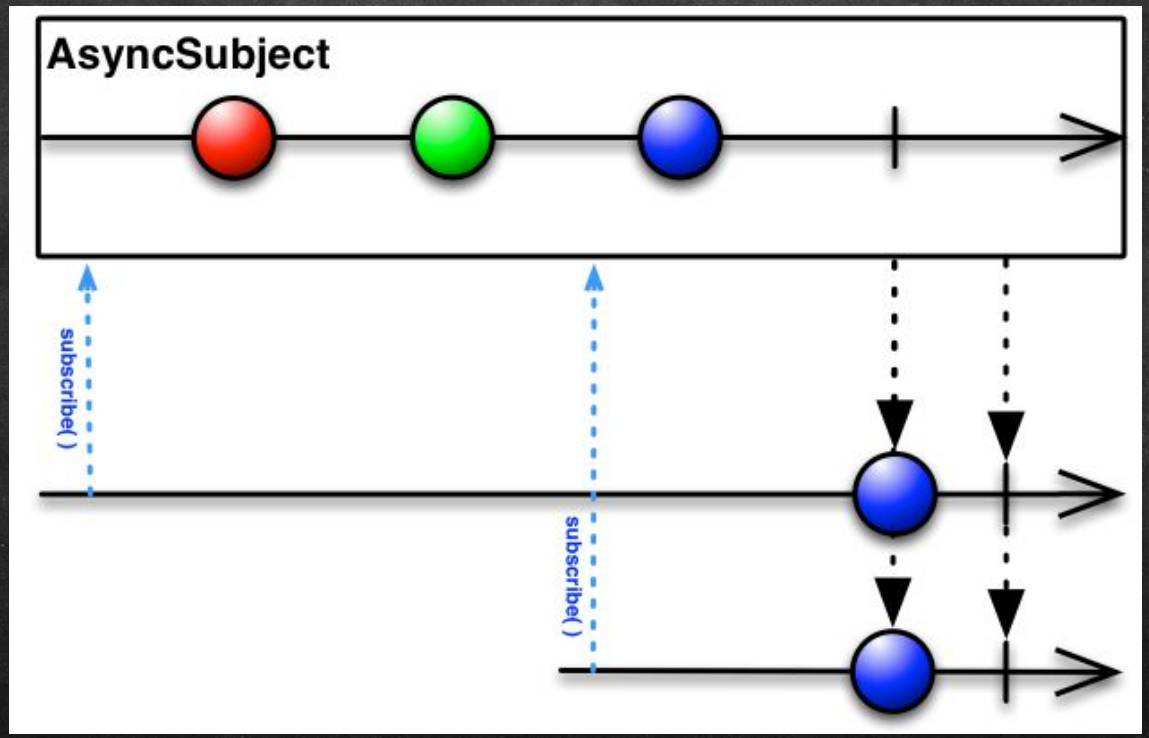
```java
BehaviorSubject<Integer> intSubject = BehaviorSubject.create();
Observable<Integer> intObserver = intSubject.asObservable();
intObserver.subscribe((v) -> System.out.println("Subscriber 1: " + v));
intSubject.onNext(1);
intSubject.onNext(2);

intObserver.subscribe((v) -> System.out.println("Subscriber 2: " + v));
intSubject.onNext(3);
intSubject.onCompleted();
```

Java

```javascript
const subject = new Rx.BehaviorSubject();
const observer = subject.asObservable();

observer.subscribe((v) => console.log('Subscriber 1: ' + v));
subject.next(1);
subject.next(2);

observer.subscribe((v) => console.log('Subscriber 2: ' + v));
subject.next(3);
subject.complete();
```

JavaScript

AsyncSubject



AsyncSubject

```java
AsyncSubject<Integer> intSubject = AsyncSubject.create();
Observable<Integer> intObserver = intSubject.asObservable();
intObserver.subscribe((v) -> System.out.println("Subscriber 1: " + v));
intSubject.onNext(1);
intSubject.onNext(2);

intObserver.subscribe((v) -> System.out.println("Subscriber 2: " + v));
intSubject.onNext(3);
intSubject.onCompleted();
```

Java

```javascript
const subject = new Rx.AsyncSubject();
const observer = subject.asObservable();

observer.subscribe((v) => console.log('Subscriber 1: ' + v));
subject.next(1);
subject.next(2);

observer.subscribe((v) => console.log('Subscriber 2: ' + v));
subject.next(3);
subject.complete();
```

JavaScript

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

Hot vs Cold Observables

```
Observable<Long> hot = Observable.interval(500,
TimeUnit.MILLISECONDS)
  .take(10).publish().refCount();
hot.subscribe((v) -> System.out.println("+ " + v));
Thread.sleep(3000);
hot.subscribe((v) -> System.out.println("++ " + v));
```

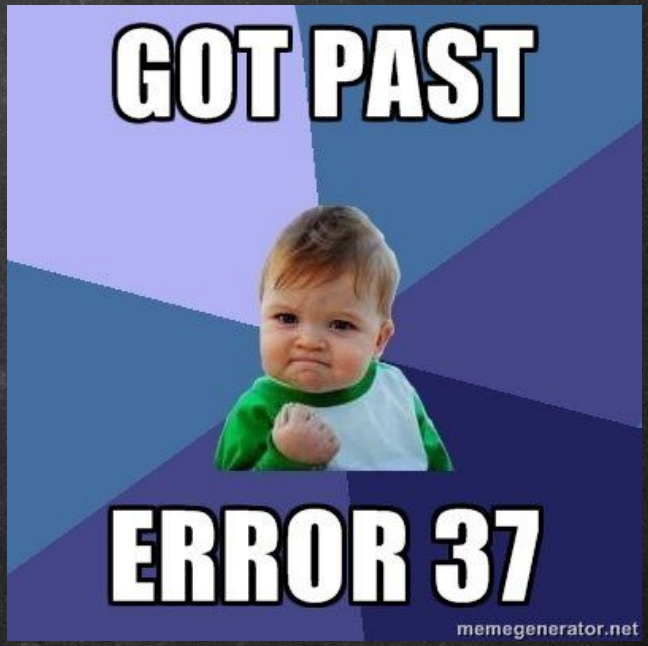
Java

```
const hot =
Rx.Observable.interval(500).take(10).publish().refCount()
;
hot.subscribe((v) => console.log('+ ' + v));
setTimeout(() => {
    hot.subscribe((v) => console.log('++ ' + v));
}, 3000);
```

JavaScript

Error handling

- Errors get emitted in a different channel
- Data channel gets closed when an error occurs
- Various recovery possibilities
  - emit default item
  - restart Observable immediately
  - restart after some time
  - switch to different Observable



Error handling

```
Observable.from(new String[] { "1", "2", "t", "4" })
  .map((n) -> Integer.parseInt(n))
  .onErrorResumeNext((error) -> Observable.from(new
Integer[]{ 10, 11 }))
  .subscribe(System.out::println, System.err::println);
```

Java

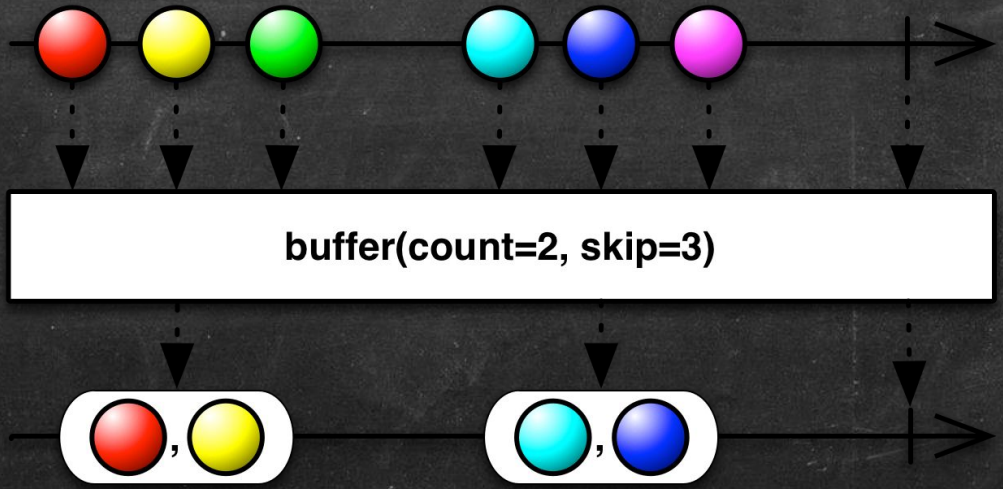
```
Rx.Observable.of('one', 'two', null, 'four')
    .map((n) => n.toUpperCase())
    .catch((error) => Rx.Observable.of('TEN', 'ELEVEN'))
    .subscribe(console.log, console.error);
```

JavaScript

CRAFTSMEN

Operators part II

buffer



CRAFTSMEN

buffer

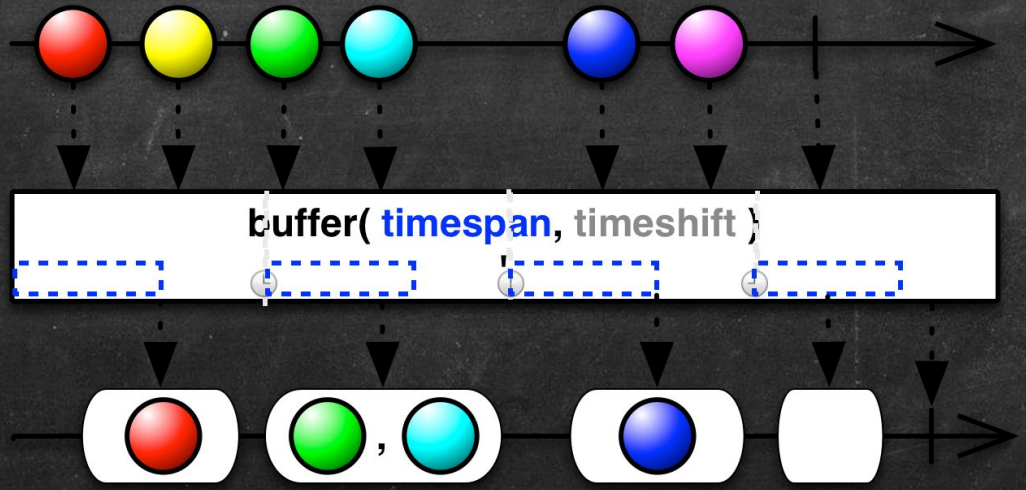
```java
Observable<String> data$ = giveMeData();

Observable<List<String>> bufferedData$ =
  data$.buffer(2, 1);

bufferedData$.subscribe(System.out::println);
```

Java

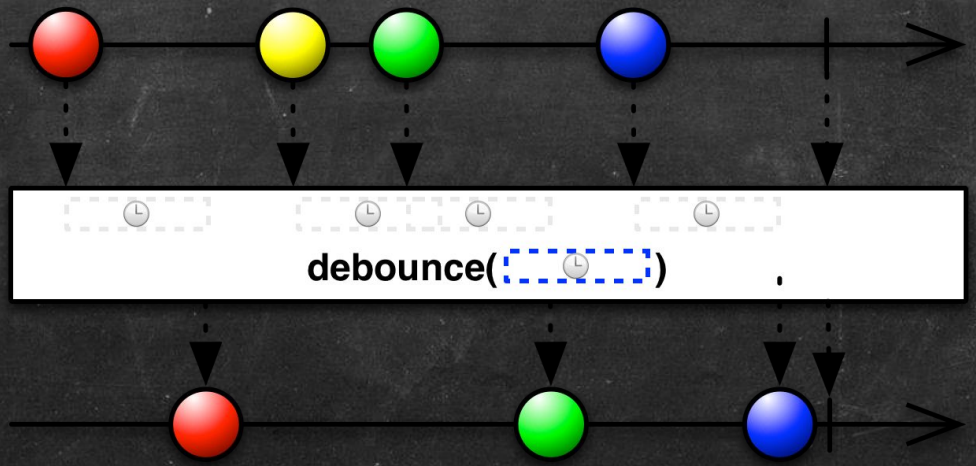
```javascript
const data$ = giveMeData();

const bufferedData$ = data$.bufferCount(2, 1);

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

JavaScript

buffer (time based)



debounce



CRAFTSMEN

debounce

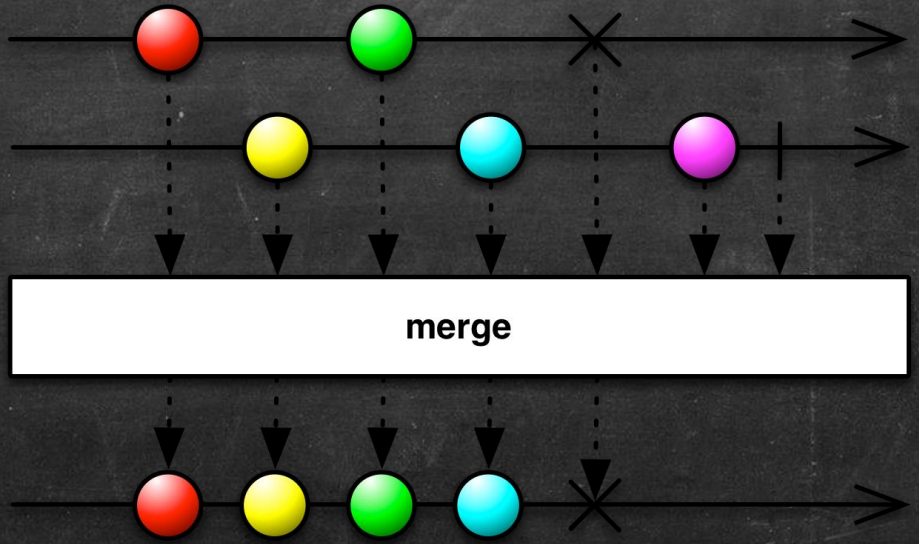
```java
Observable<String> data$ = giveMeData();

Observable<String> debouncedData$ =
  data$.debounce(1500, TimeUnit.MILLISECONDS);

debouncedData$.subscribe(System.out::println);
```

Java

```javascript
const data$ = giveMeData();

const debouncedData$ = data$.debounceTime(1500);

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

JavaScript

merge



CRAFTSMEN

merge

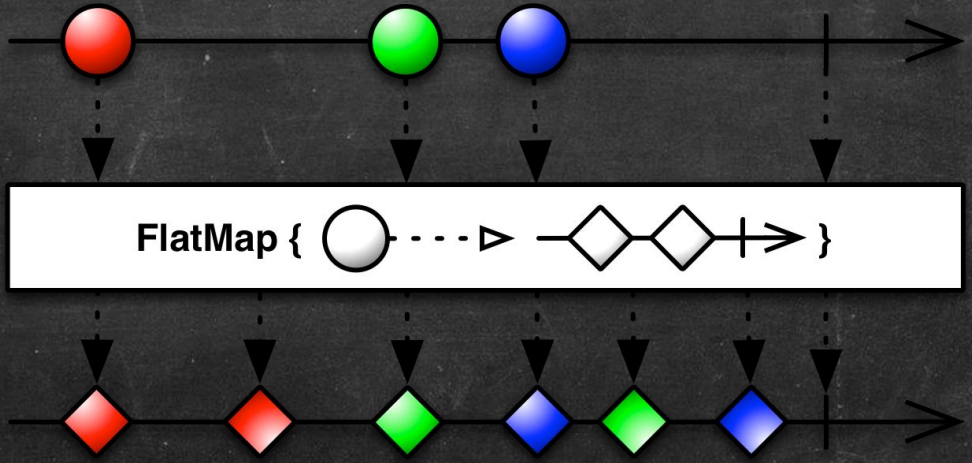
```java
Observable<String> data$ = giveMeData();
Observable<String> moreData$ = giveMeMoreData();

Observable<String> mergedData$ =
  Observable.merge(data$, moreData$);

mergedData$.subscribe(System.out::println);
```

Java

```javascript
const data$ = giveMeData();
const moreData$ = giveMeMoreData();

const mergedData$ =
  Rx.Observable.merge(data$, moreData$);

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

JavaScript

mergeMap (a.k.a. flatMap)



mergeMap (a.k.a. flatMap)

CRAFTSMEN

mergeMap (a.k.a. flatMap)

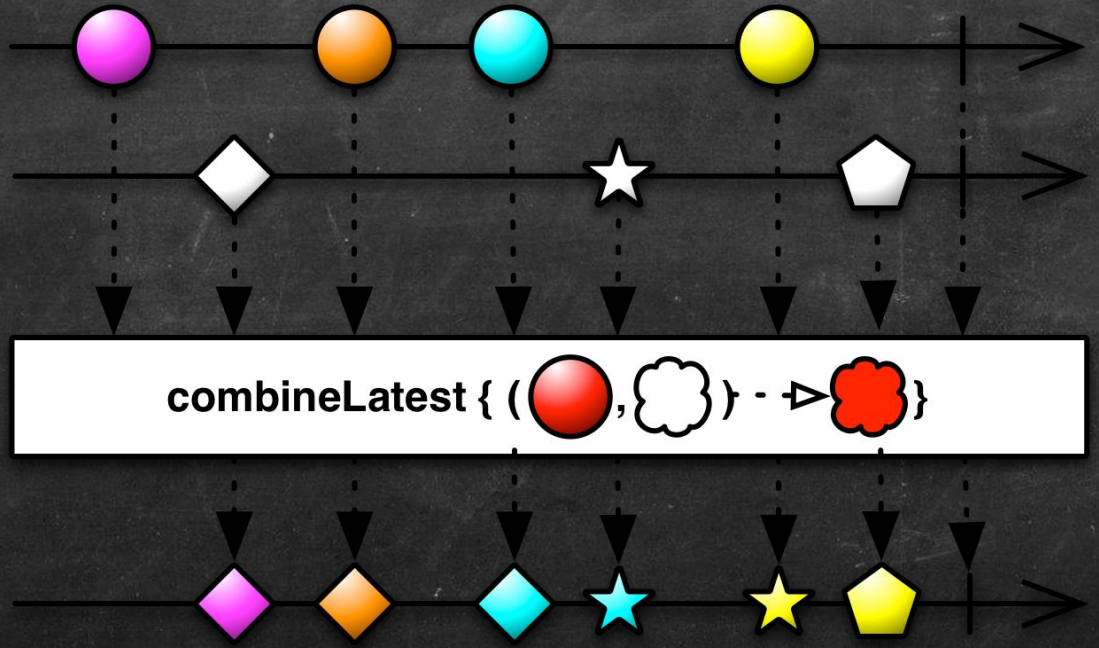
```java
Observable<String> data$ = Observable.range(1, 5)
  .flatMap((i) ->
    Observable.range(1, i)
      .map((j) -> i + "." + j)
  );

data$.subscribe(System.out::println);
```

Java

```javascript
const data$ = Rx.Observable.range(1, 5)
  .mergeMap((i) =>
    Rx.Observable.range(1, i)
      .map((j) => i + '.' + j)
  );

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

JavaScript

combineLatest



CRAFTSMEN

combineLatest

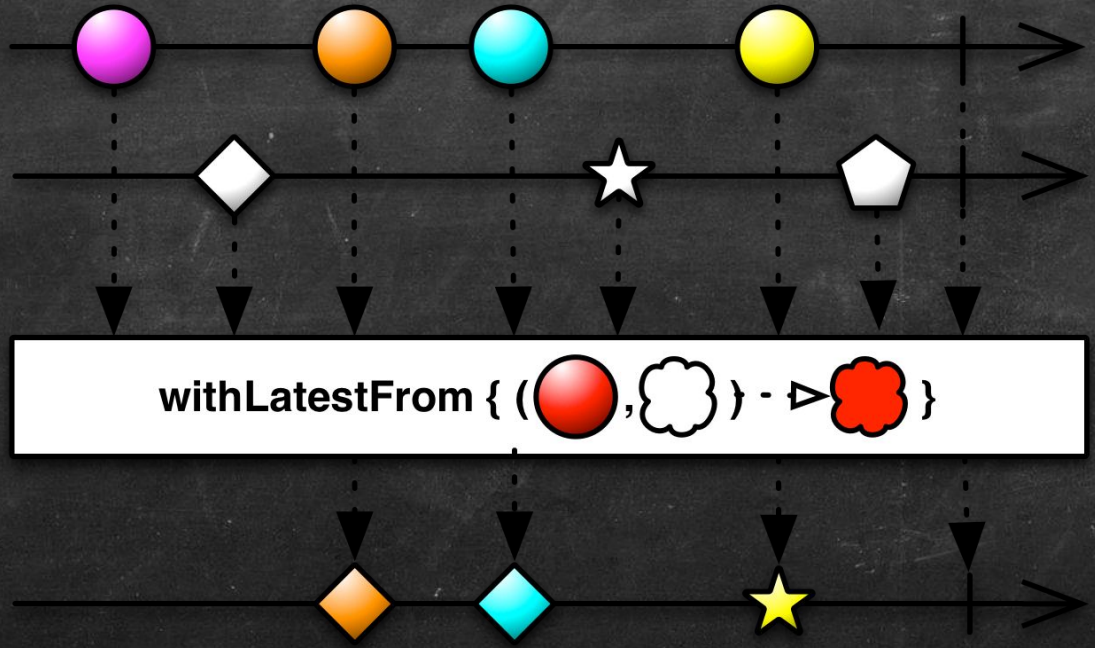
```java
Observable<String> data$ = giveMeData();
Observable<String> moreData$ = giveMeMoreData();

Observable<String> combinedData$ =
  Observable.combineLatest(
    data$, moreData$, (a, b) -> a + "-" + b);

combinedData$.subscribe(System.out::println);
```

Java

```javascript
const data$ = giveMeData();
const moreData$ = giveMeMoreData();

const combinedData$ =
  Rx.Observable.combineLatest(
    data$, moreData$, (a, b) => a + '-' + b);

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

JavaScript

withLatestFrom



CRAFTSMEN

withLatestFrom

```java
Observable<String> data$ = giveMeData();
Observable<String> moreData$ = giveMeMoreData();

Observable<String> combinedData$ =
  data$.withLatestFrom(
    moreData$, (a, b) -> a + "-" + b);

combinedData$.subscribe(System.out::println);
```

Java

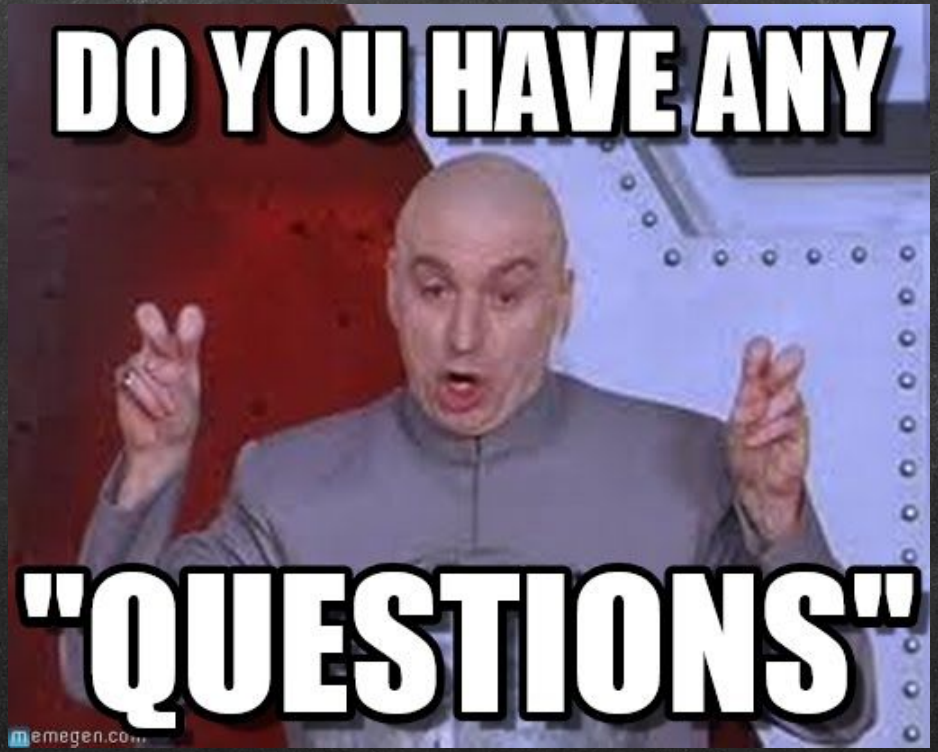
```javascript
const data$ = giveMeData();
const moreData$ = giveMeMoreData();

const combinedData$ =
  data$.withLatestFrom(
    moreData$, (a, b) => a + '-' + b);

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

JavaScript

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$ = x$.combineLatest(y$,
   (x, y) => x * y);

z$.subscribe(console.log);

y$.next(3);
```


DO YOU HAVE ANY
"QUESTIONS"

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

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
  - First-class citizen
  - Composable (through operators)

Stuff we haven’t covered

- Unsubscribing / disposing observables
- Many more operators
  - Different flavours of presented operators
- Backpressure
- Schedulers and Threading
- Testing and debugging
- Check README.md for more material

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


CRAFTSMEN
FIN