REACTIVE PROGRAMMING USING RXJS

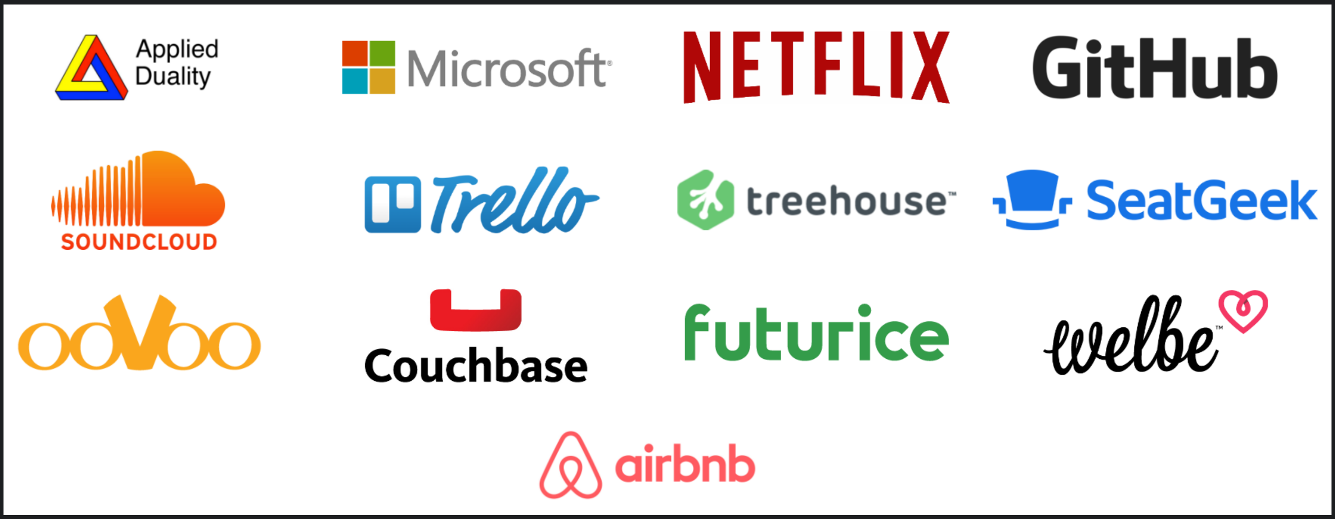
OBSERVER & OBSERVABLE

	Single	**Multiple**
Pull	Function	Iterator
Push	Promise	Observable

Promises are designed to handle a single response value

Observable are designed to handle an **infinity** of response values

WE USE REACTIVEX



Supported by Java, JavaScript, C#, Scala, Python, etc ...

GETTING STARTED

PROMISE / FUTURE

```
new Promise((resolve, reject) => {
    resolve(42);
}).then((x) => console.log(x));

Promise
    .resolve(42)
    .then((x) => console.log(x));
```

OBSERVABLE

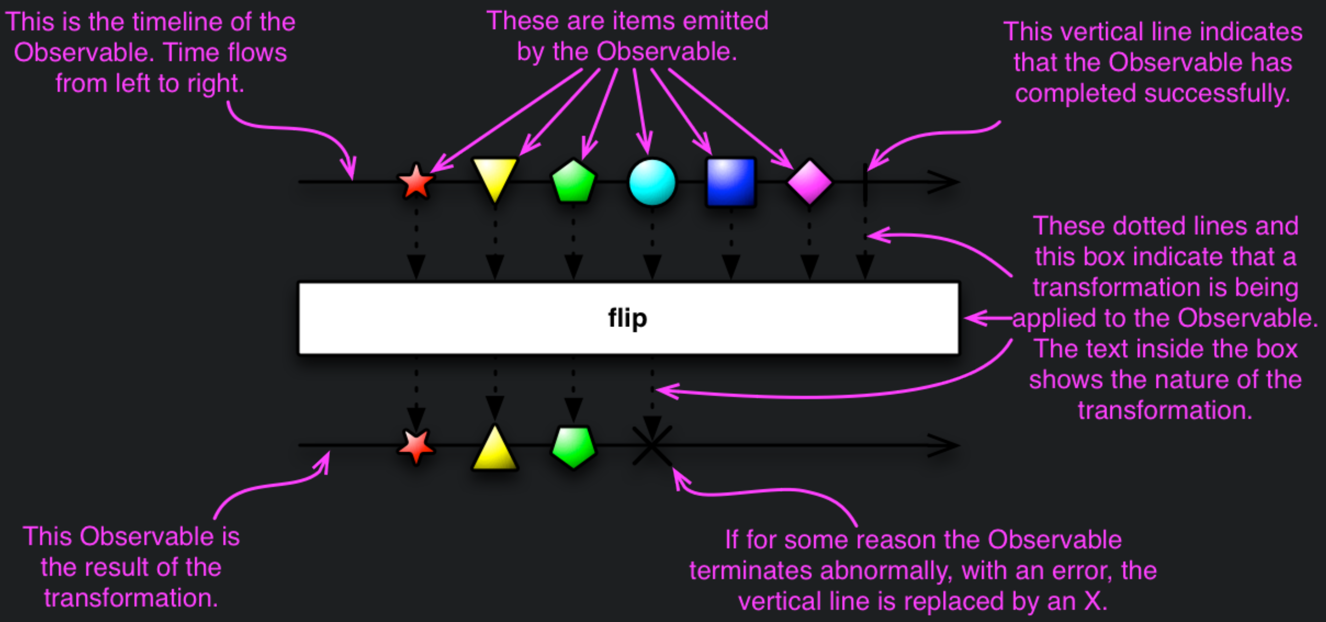
```
import * as Rx from 'rxjs/Rx'

Rx.Observable
  .create((observer) => observer.next(42))
  .subscribe((x) => console.log(x));

Rx.Observable
  .of(42)
  .subscribe((x) => console.log(x));
```

SUBSCRIPTION

```
import * as Rx from 'rxjs/Rx'

const source = Rx.Observable.of(42);

//Process some code ...

source.subscribe(x => console.log(x));
source.subscribe(x => console.log(x + 2));
```

STREAM & FUNCTIONAL PROGRAMMING

OPERATORS


This is the timeline of the Observable. Time flows from left to right.
These are items emitted by the Observable.
This vertical line indicates that the Observable has completed successfully.
These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.
flip
This Observable is the result of the transformation.
If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

OPERATOR

MAP

OPERATOR

FILTER

OPERATOR

REDUCE

OPERATOR

FIND

OPERATOR

MAX

OPERATOR

SUM

EXAMPLE

```
import * as Rx from 'rxjs/Rx'

const data = [
    { name: 'Bob', age: 25 },
    { name: 'Alice', age: 31 }
];

const source = Rx.Observable.from(data);
const sample = source.take(1000);
sample
    .filter(person => person.age >= 30)
    .reduce((acc, person) => acc + 1, 0)
    .map(count => `${count} persons`)
    .subscribe(console.log);
sample
    .max(person => person.age)
    .map(p => `The oldest is ${p.name}`)
    .subscribe(console.log);
```

SUBSCRIBE TO STREAM

OPERATOR

SCAN

OPERATOR

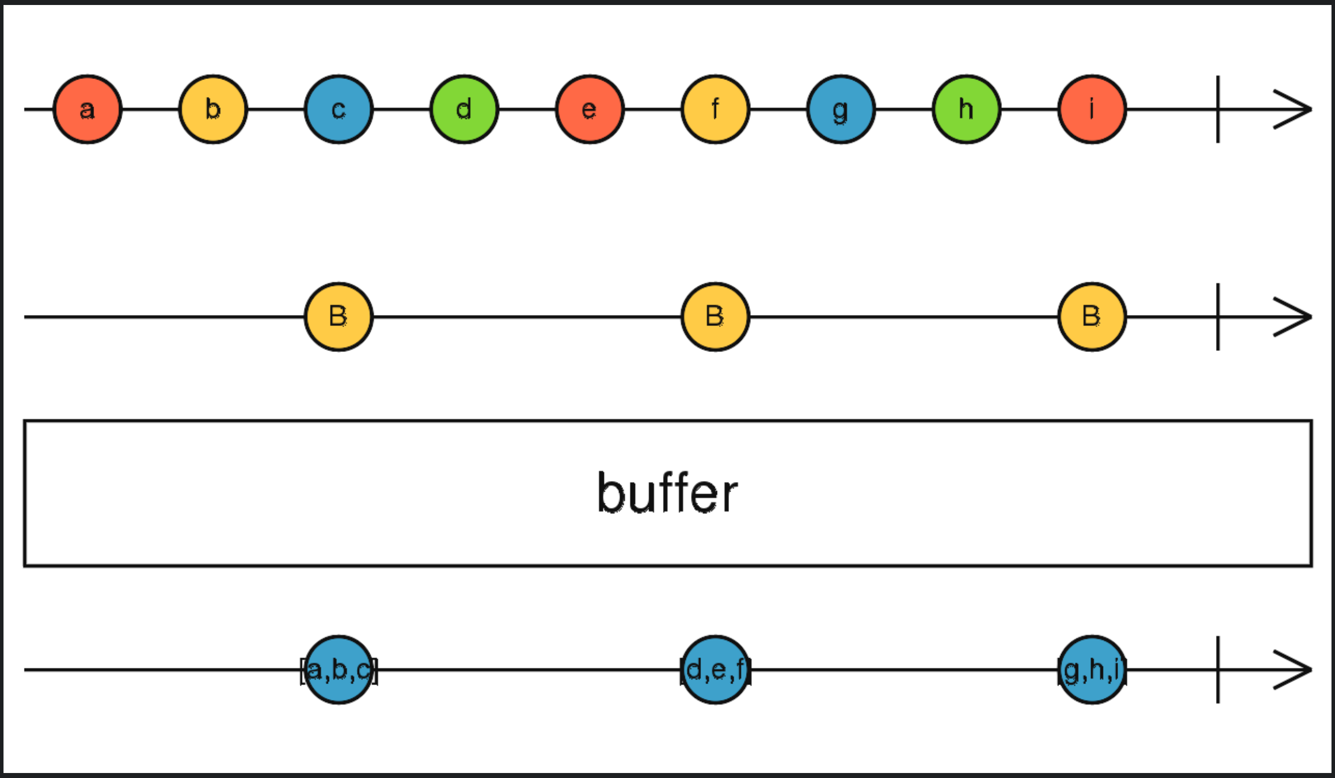
DELAY

OPERATOR

DEBOUNCE

OPERATOR

BUFFER



COMBINE OBSERVABLE

OPERATOR

MERGE

OPERATOR

CONCAT

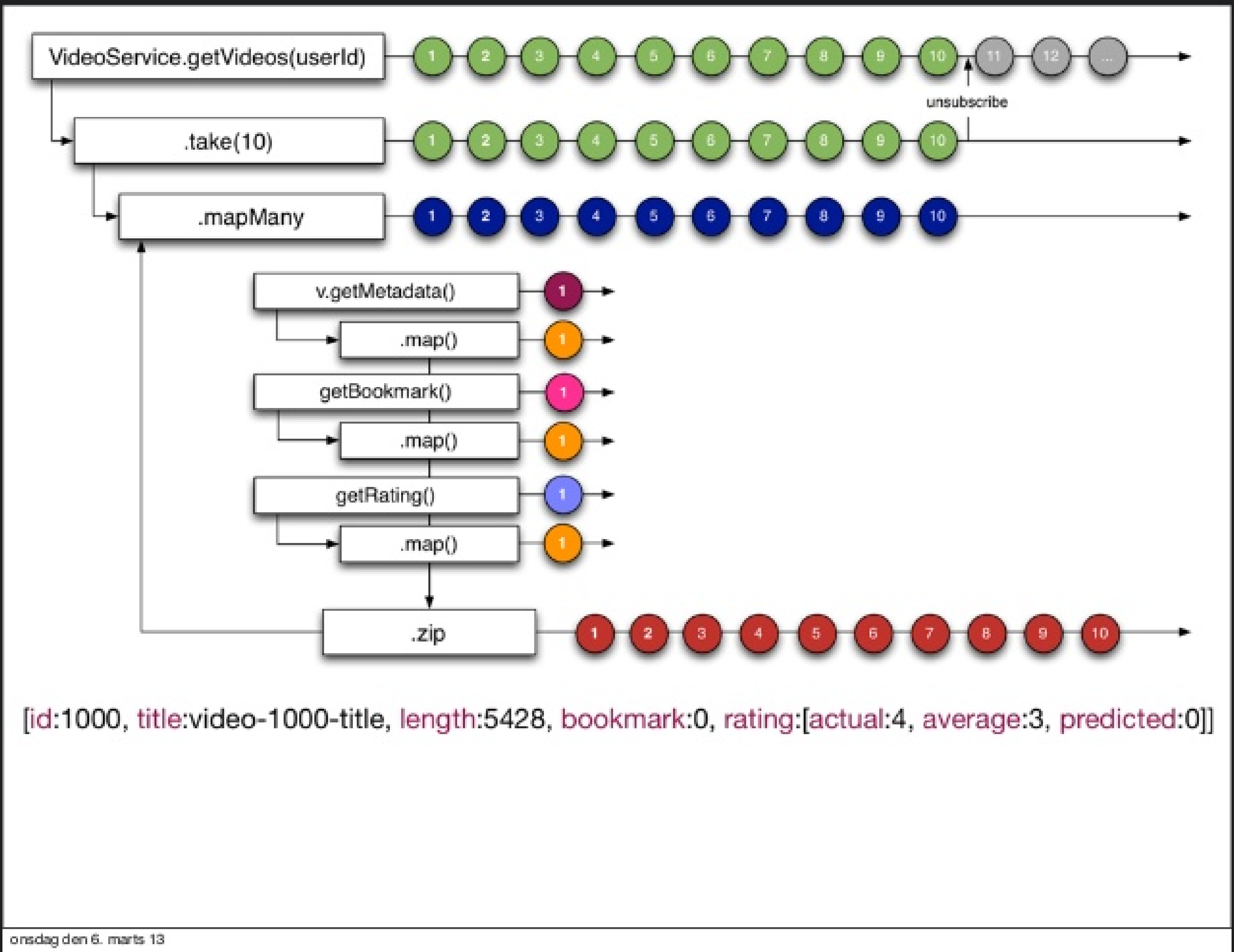
OPERATOR

ZIP

```
import * as Rx from 'rxjs/Rx'
const b = document.querySelector('#send');
const b2 = document.querySelector('#unsub');
const source = Rx.Observable.fromEvent(b,'click');
const unsub = Rx.Observable.fromEvent(b2,'click');
source.map(m => `Pos: ${m.clientX}, ${m.clientY}`)
 .subscribe(console.log);
source.scan((acc, _) => acc + 1, 0)
 .map(c => `Count: ${c}`)
 .subscribe(console.log);

const personStream = Rx.Observable
      .interval(700)
      .map(i => ({age: i, name: `Bob ${i}`}));
const everySeconds = personStream
      .buffer(Rx.Observable.interval(1000));
const total = everySeconds.map(p => p.map(a => a.age)
        .reduce((a, b) => a + b, 0)
        );
const size = everySeconds.map(p => p.length);
const zipped = Rx.Observable
 .zip(total, size)
 .map(([total, size]) => total / size)
 .map(average => `Average: ${average}`)
 .subscribe(console.log);
unsub.subscribe(_ => zipped.unsubscribe());
```

EXAMPLE: FETCH FILMS WITH NETFLIX

EXAMPLE: NETFLIX



[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

onsdag den 6. marts 13

SOME CONCEPTS

PURE FUNCTIONAL

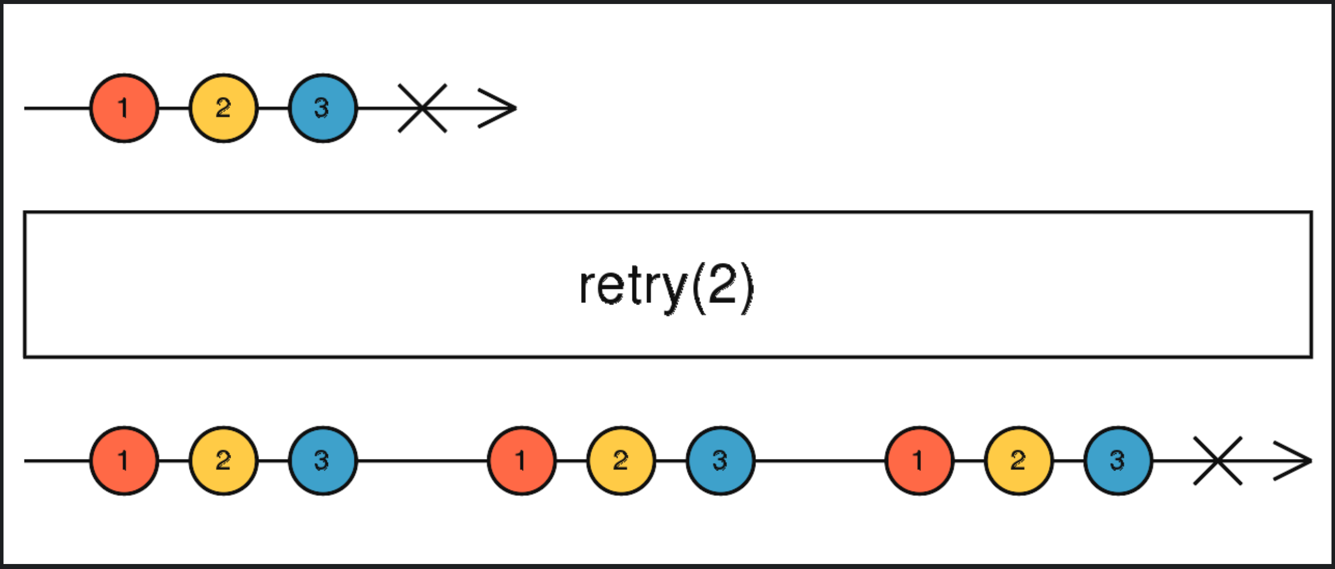
Avoid stateful programs, using clean input/output functions over observable streams.

LAZINESS

Only calls to "subscribe" trigger the evaluation (like action on Spark)

ASYNC ERROR HANDLING

Traditional try/catch is powerless for errors in asynchronous computations, but ReactiveX is equipped with proper mechanisms for handling errors.



EXAMPLE

```
import * as Rx from 'rxjs/Rx';

const source = Rx.Observable.interval(1000);

const predicate = (val) => {
    return val >= 3
    ? Rx.Observable.throw('Error')
    : Rx.Observable.of(val);
}

const test = source
  .flatMap(predicate)
  .retry(2);

test.subscribe(
     val => console.log(val),
     err => console.log(err)
);
```

CONCURRENCY MADE EASY

Observables allow to abstract away low-level threading, synchronization, and concurrency issues.

SUBJECTS

SIMPLE SUBJECT

A Subject is a bridge that acts both as an observer and as an Observable

EXAMPLE

```
import * as Rx from 'rxjs/Rx'

const subject = new Rx.Subject();

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});

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

ASYNC SUBJECT

The AsyncSubject is a variant where only the last value of the Observable execution is sent to its observers, and only when the execution completes

EXAMPLE

```
import * as Rx from 'rxjs/Rx'

const subject = new Rx.AsyncSubject();

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});

subject.next(1);
subject.next(2);
subject.next(3);
subject.next(4);

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});

subject.next(5);
subject.complete();
```

REPLAY SUBJECT

A ReplaySubject records multiple values from the Observable execution and replays them to new subscribers.

EXAMPLE

```
import * as Rx from 'rxjs/Rx'

const subject = new Rx.ReplaySubject(3);

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});

subject.next(1);
subject.next(2);
subject.next(3);
subject.next(4);

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});

subject.next(5);
```

BEHAVIOR SUBJECT

Whenever a new Observer subscribes, it will immediately receive the "current value" from the BehaviorSubject. BehaviorSubjects are useful for representing "values over time". For instance, an event stream of birthdays is a Subject, but the stream of a person's age would be a BehaviorSubject.

EXAMPLE

```
import * as Rx from 'rxjs/Rx'

const subject = new Rx.BehaviorSubject(0);

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});

subject.next(1);
subject.next(2);

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});

subject.next(3);
```

TP: CHAT