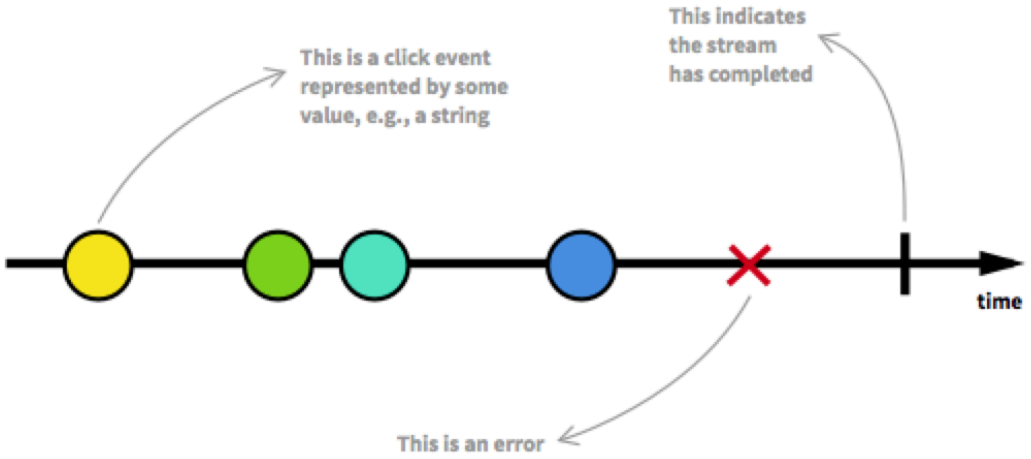
Programación Funcional Reactiva

- Programación en base a streams de datos asincrónicos
- Streams pueden ser de cualquier cosa (no solo eventos)
- Toolbox de funciones permite crear, filtrar y combinar estos streams
- Uno o varios streams puede ser el input de otro

Streams

- Sequencia de eventos ordenados en el tiempo
- Puede emitir un valor, un error o un "completed"
- Los eventos son capturados en forma asíncrona definiendo funciones para cuando se emiten valores, errores y cierre.
- Patrón Observer
  - Escuchar al stream se le llama "subscribirse"
  - El Stream es el sujeto u observable

Símbología (Marble Diagram)



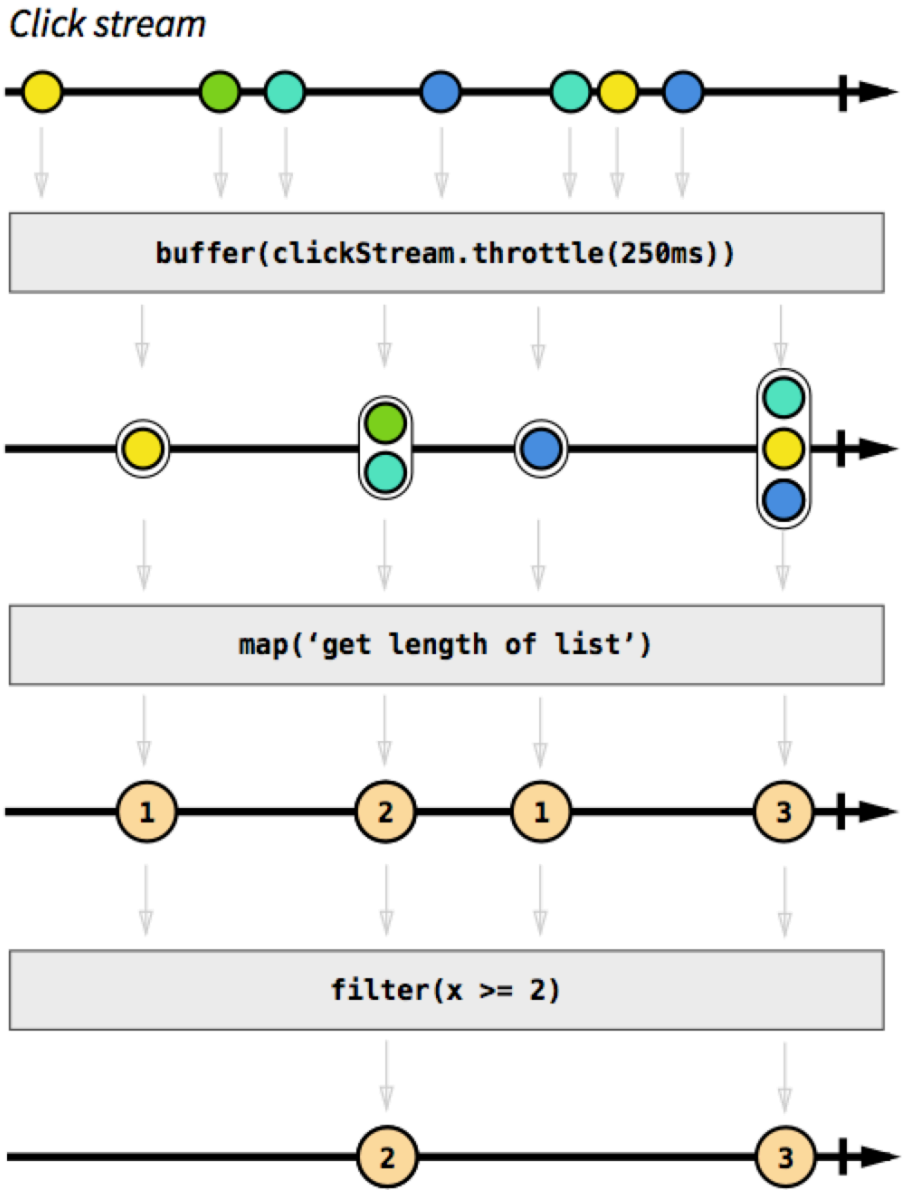

```
--a---b-c---d---X---|->

a, b, c, d are emitted values
X is an error
| is the 'completed' signal
---> is the timeline
```

Ejemplo muy simple

- Contar el número de clicks

```
clickStream: ---c----c--c----c------c-->
               vvvvv map(c becomes 1) vvvv
               ---1----1--1----1------1-->
               vvvvvvvvv scan(+) vvvvvvvvv
counterStream: ---1----2--3----4------5-->
```

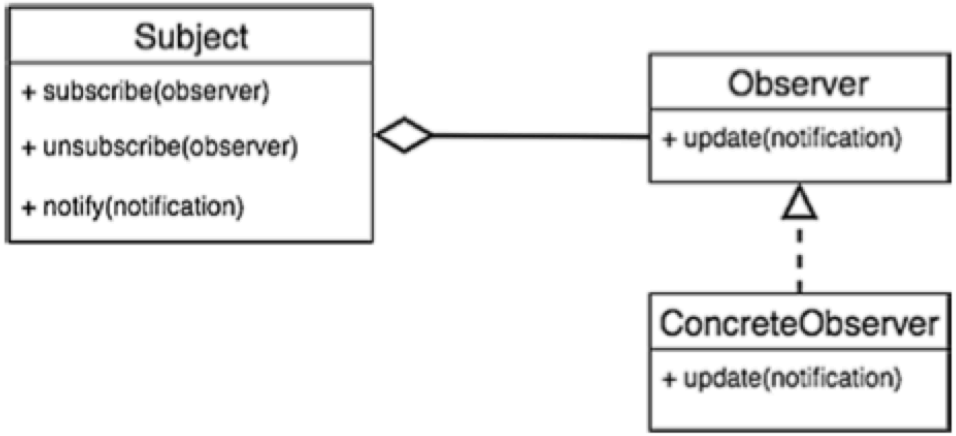
Contando dobles y triples clicks



RxJS

- Librería JS para programación reactiva (implementación de ReactiveX)
- Basada en la idea de observables
- Como Lodash pero para eventos
- Conceptos
  - observable: colección de valores o eventos futuros
  - observer: colección de callbacks que sabe como escuchar a los valores de observable
  - operator: método de observable que crea un nuevo observable sin modificar el primero
- Otras librerías relevantes:
  - XStream (streams and listeners)
  - Most.js
  - IxJS
  - Kefir
  - Bacon

El patrón observer



- El sujeto mantiene una lista de observadores que se subscriben para ser notificados de cambios

Ejemplo muy simple

```
const observable = Observable.range(1, 10);
const observer = observable.subscribe(onData, onError, onComplete);
function onData(value){
        console.log(value);
}
function onError(err){
        console.error(err);
}
function onComplete(){
        console.log("stream complete!");
}

1
2
3
4
5
6
7
8
9
10
stream completed
```

Variación ...

```
import Rx from "rxjs"
const data = [1,2,10,1,3,9,6,13,11,10,10,3,19,18,17,15,4,8,4];

const onData = (value) => console.log(`current sum is ${value}`);
const onError = _ => console.log("stream error");
const onComplete = _ => console.log("stream completed");

const obs = Rx.Observable.from(data)
                  .filter(value => value % 2 === 0)
                  .distinct()
                  .reduce((acc, value) => acc + value);

obs.subscribe(onData, onError, onComplete);



current sum is 48
stream completed
```

Creación de observables desde ...

- valores simples - of(arg)
- desde arrays - from(iterable)
- desde promesas - fromPromise(promise)
- desde eventos - fromEvent(element, eventName)

El Observer

- representa el lado del consumidor
- reacciona a cambios producidos en el observable
- define tres funciones
  - next - se llama cada vez que hay un nuevo evento
  - error - se llama en caso de una excepción
  - complete - se llama cuando todos los eventos han sido procesados

- se pueden entregar las 3 funciones en el subscribe
- puede crearse solo con subscribe y pasandole solo una función que va a corresponder al next

Observers

```
var observer = {
  next: x => console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
  complete: () => console.log('Observer got a complete notification'),
};
```

```
observable.subscribe(
  x => console.log('Observer got a next value: ' + x),
  err => console.error('Observer got an error: ' + err),
  () => console.log('Observer got a complete notification')
);
```

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

Tipos de Operadores

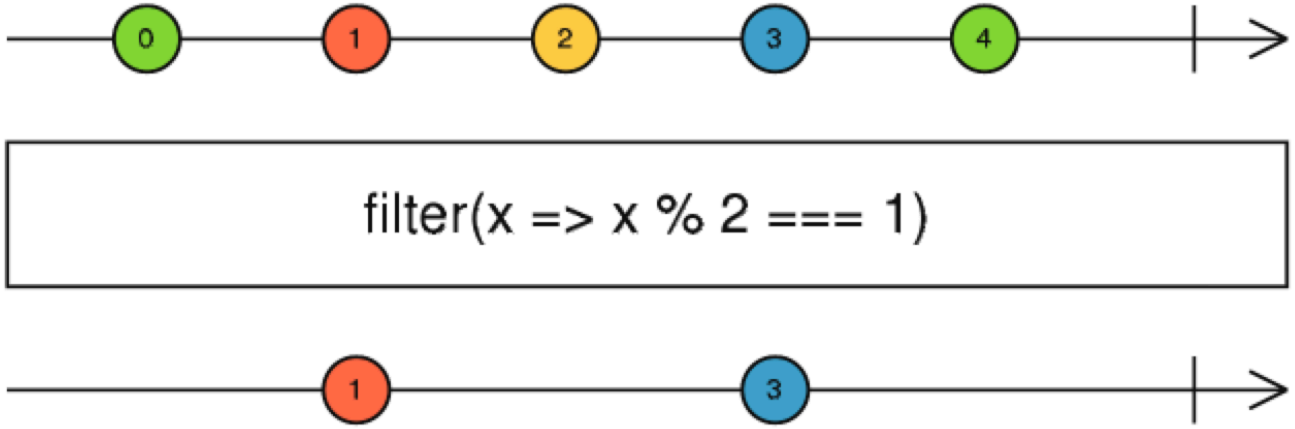
- Creation: create, range…
- Transformation: buffer, map…
- Filtering: distinct, take…
- Combination: concat, merge…
- Multicasting: publish, share…
- Error handling: catch, retry…
- Utility: do, delay…
- Conditional: isEmpty, find…
- Mathematical: count, reduce…

Operadores muy Usados

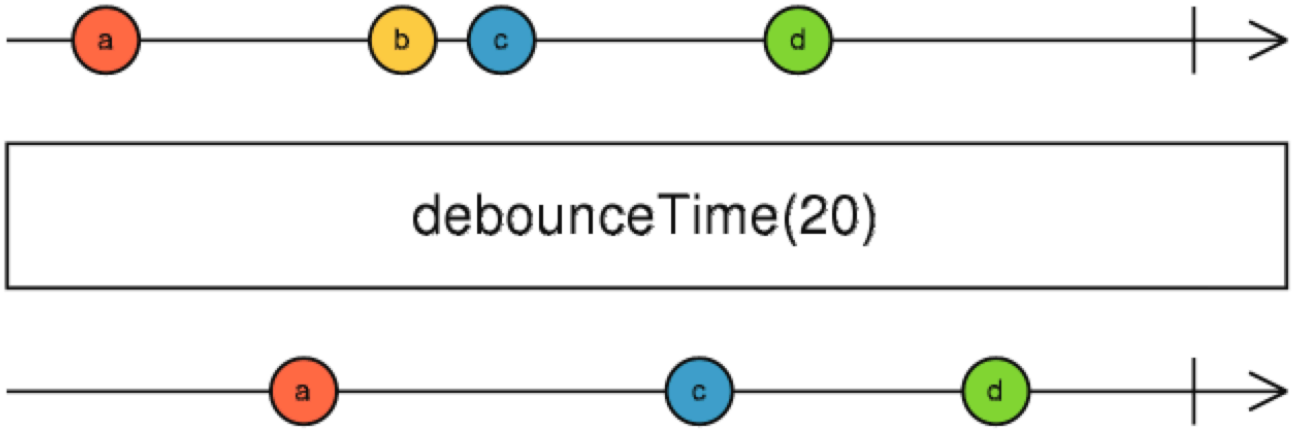
- from - convierte cualquier cosa en observable
- fromEvent (element, 'click') - crea un observable para eventos del DOM
- fromPromise - devuelve un observable que emite el valor de la promesa cuando resuelve

```
var result = Rx.Observable.fromPromise(fetch('http://myserver.com'));
result.subscribe(x => console.log(x), e =>console.error(e));
```

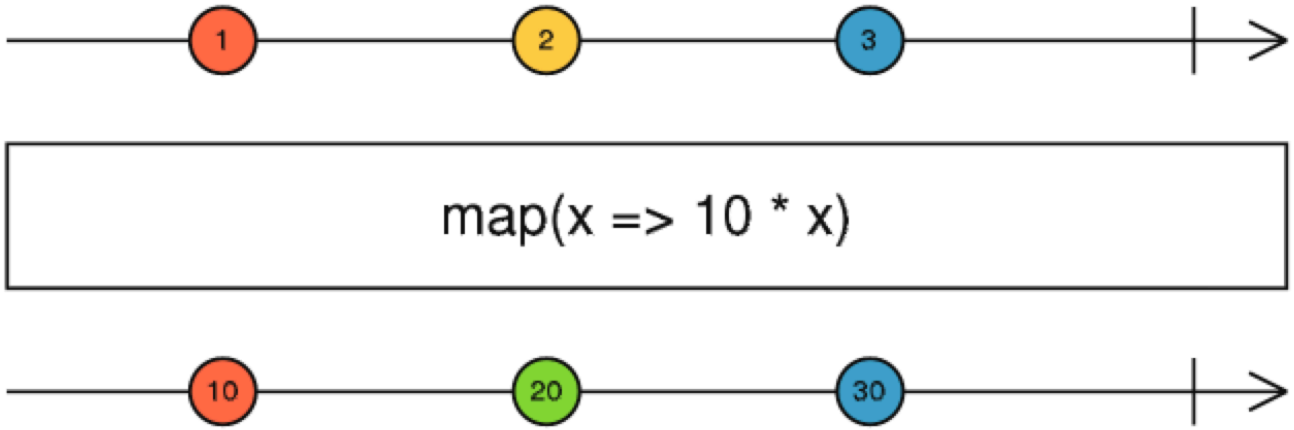
Filter



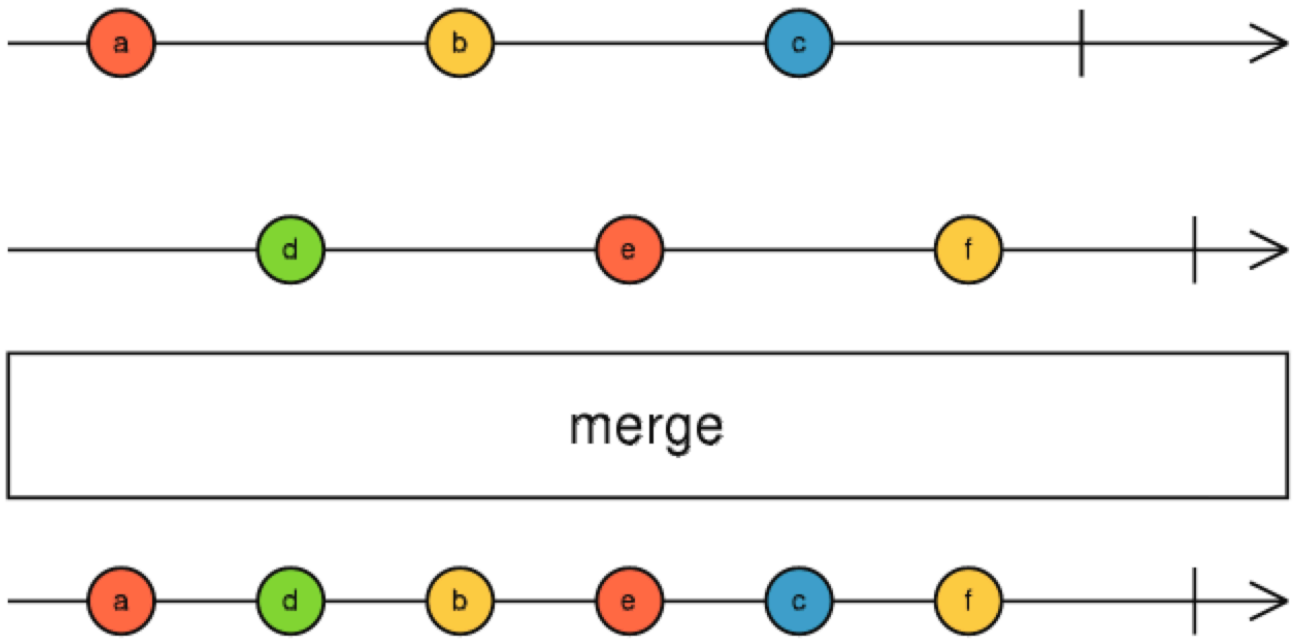
DebounceTime



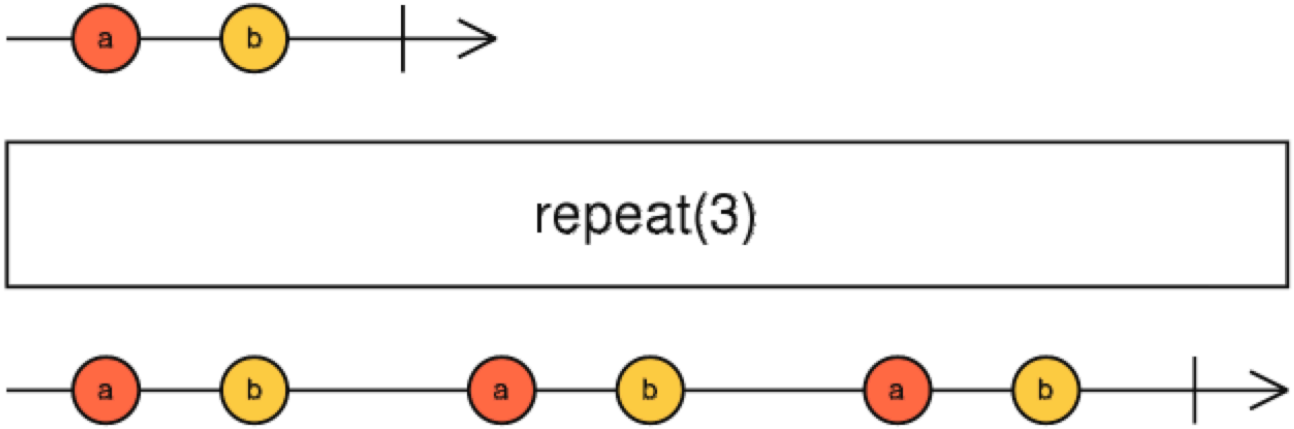
map



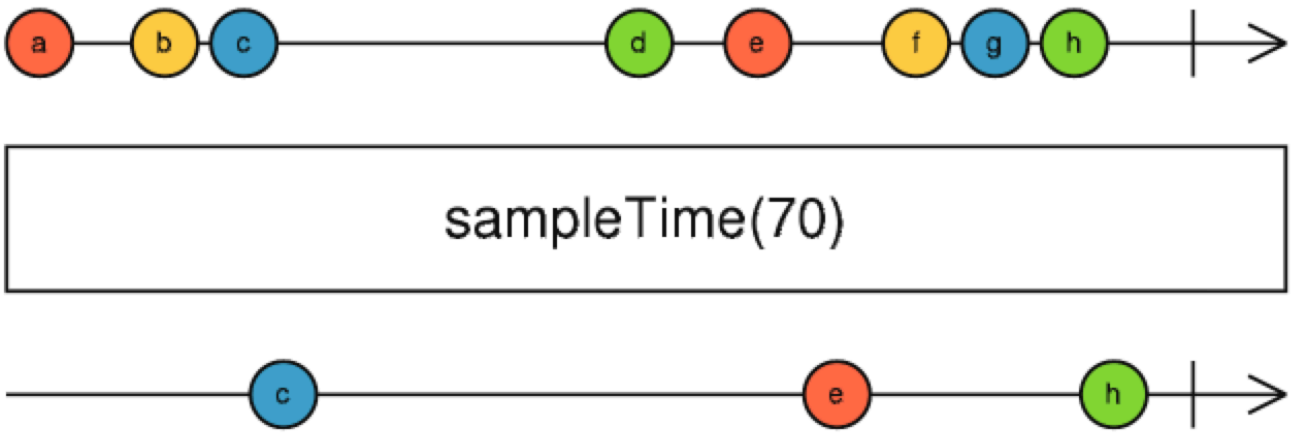
merge



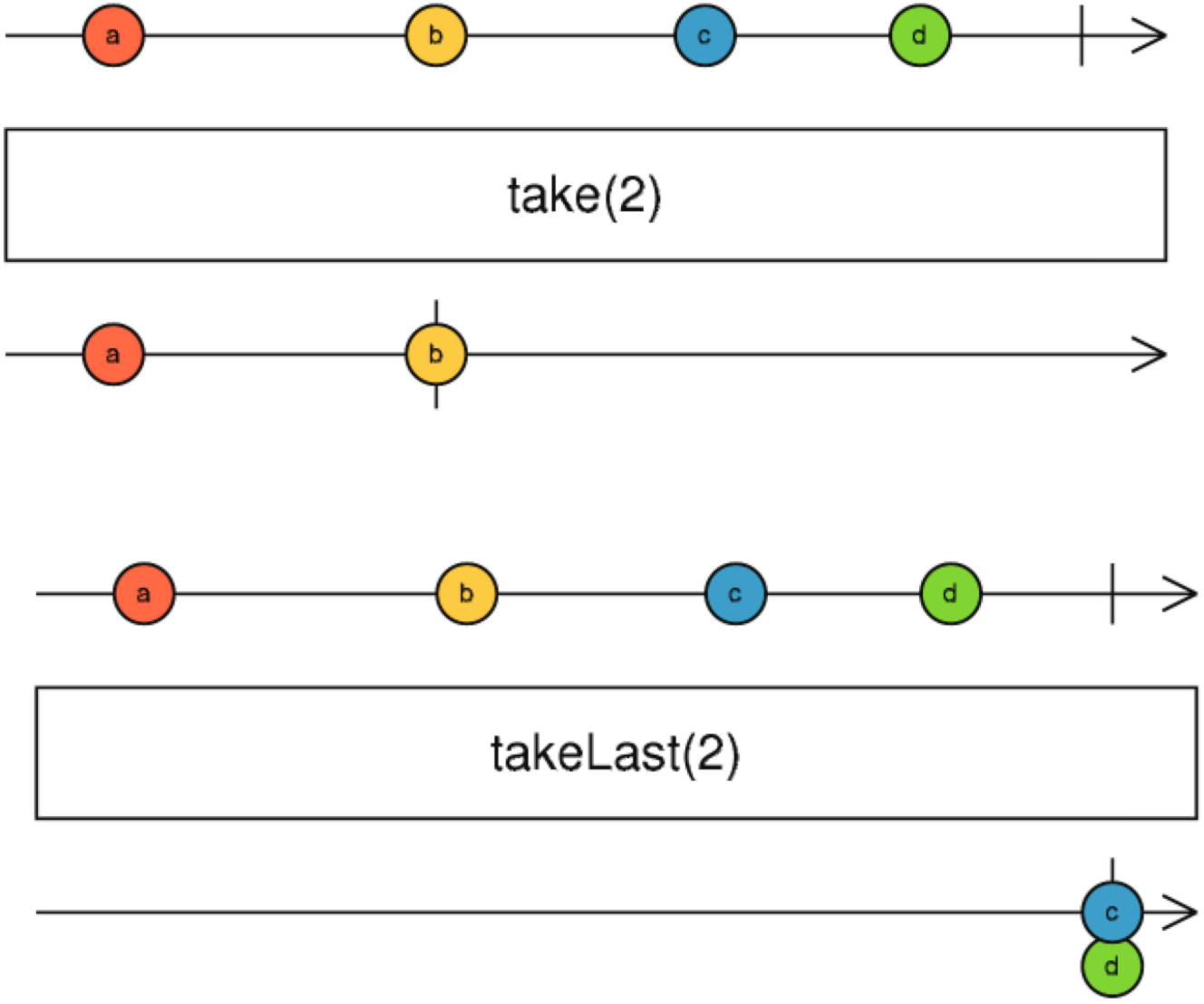
repeat



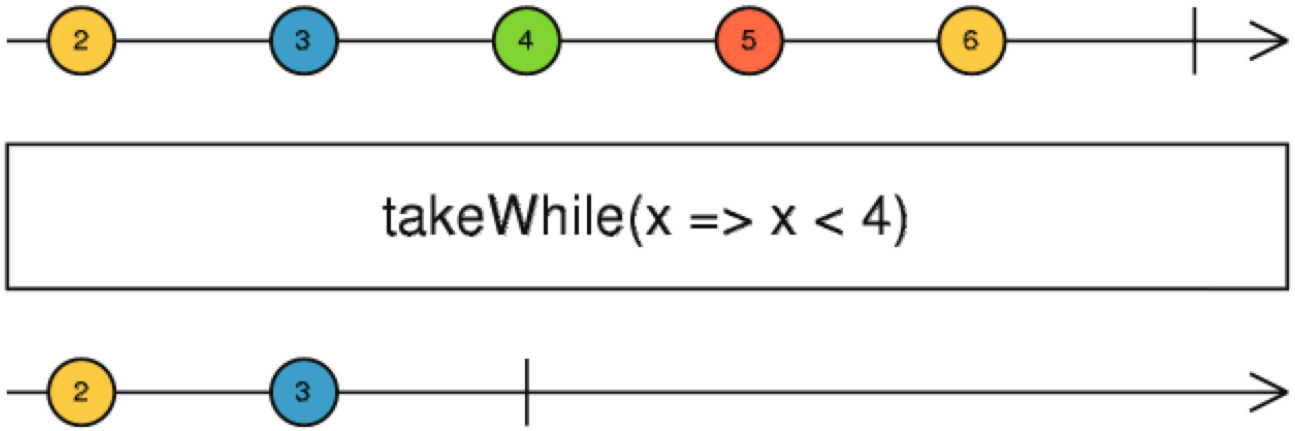
sampleTime



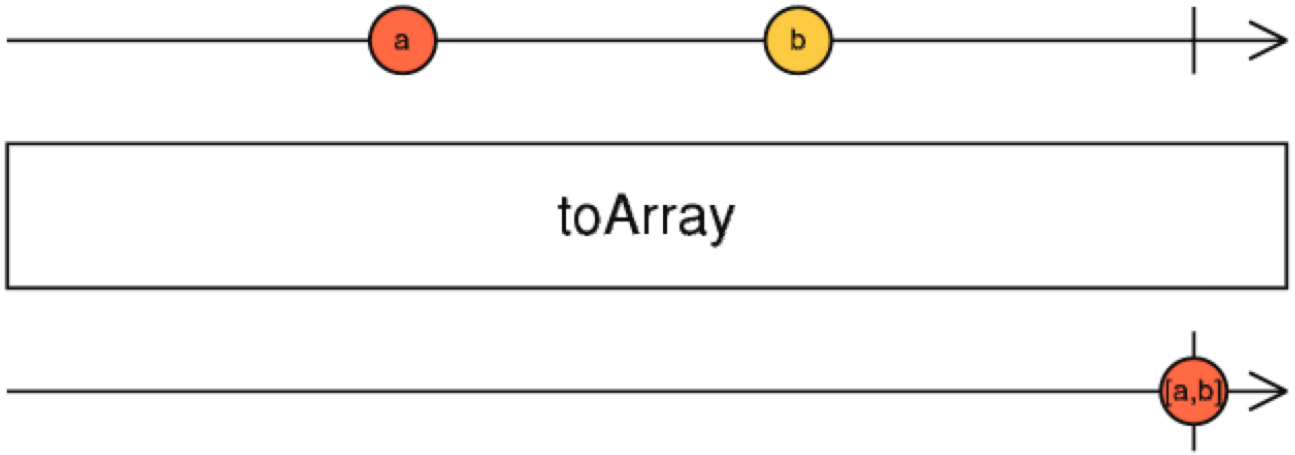
take, takeLast



takeWhile



toArray



Operadores ad hoc

- Multiplica por 10

```
function multiplyByTen(input) {
   // We return another Observable
   return Observable.create((observer) => {
      // We subscribe to the input observable to define
      // next, error and complete functions based on it
      input.subscribe(
         (value) => observer.next(value*10), // next
         (error) => observer.error(error), // error
         () => observer.complete() // complete
      );
   });
}

const input = Observable.from([1, 2, 3, 4]);
const output = multiplyByTen(input);
output.subscribe(x => console.log(x)); // 10, 20, 30, 40
```

Ejemplo Web

- Autocomplete

```
// We are creating an Observable from a DOM event
const inputStream = Observable.fromEvent(input, 'keyup')
    .map(e => e.target.value)
    .filter(value => value.length > 2)
    .distinctUntilChanged()
    .debounceTime(500)
    .mergeMap(word => this.http.get('...word...'))
    .retry(2)
    .subscribe(res => console.log(res))
```

- Listen keyboard presses.
- Extract the value associated with the input on the key event.
- Filter the values shorter than 3 chars.
- Only emit when the current value is different from the previous one.
- Discard emitted values that take less than the 500ms between output.
- Make an external http petition and get a result.
- In case of failure, retry a maximum of two times.
- Finally, we react to this result, printing it through the console.

Hot and Cold Observables

- cold observable, unicast, comienza a emitir tan pronto alguien se subscribe
  - el producer vive en el observable (se crea un nuevo producer cada vez que algien subscribe)
- hot observable, multicast, pueden emitir eventos antes de que alguien subscriba
  - el producer es uno solo y compartido entre los observers

Ejemplo Cold

```
import Rx from "rxjs";
const source = Rx.Observable.interval(2000).startWith(123)
source.subscribe(value => console.log("first observer", value))
setTimeout(_ =>{
    source.subscribe(value => console.log("second observer", value))
}, 5000);
setTimeout(_ =>{
    source.subscribe(value => console.log("third observer", value))
}, 8000)
```

```
first observer 123
first observer 0
first observer 1
second observer 123
first observer 2
second observer 0
third observer 123
first observer 3
second observer 1
third observer 0
first observer 4
second observer 2
third observer 1
first observer 5
second observer 3
third observer 2
```

Ejemplo Hot

```
import Rx from "rxjs";

const source = Rx.Observable.interval(2000)
               .startWith(123)
               .publish()
               .refCount();
source.subscribe(value => console.log("first observer", value))
setTimeout(_ =>{
        source.subscribe(value => console.log("second observer", value))
}, 5000);
setTimeout(_ =>{
        source.subscribe(value => console.log("third observer", value))
}, 8000)
```

```
first observer 123
first observer 0
first observer 1
first observer 2
second observer 2
first observer 3
second observer 3
third observer 3
first observer 4
second observer 4
third observer 4
first observer 5
second observer 5
third observer 5
first observer 6
second observer 6
third observer 6
```

Ejemplo

```
import Rx from "rxjs";
const URL = "https://jsonplaceholder.typicode.com/users";

const simplifyUserData = (user) => {
          return {
                  name: user.name,
                  email: user.email,
                  website: user.website
          }
}
const intervalObs = Rx.Observable.interval(1000)
                  .take(2)
                  .mergeMap(_ => fetch(URL))
                  .mergeMap(data => data.json())
                  .mergeAll()
                  .map(simplifyUserData)
intervalObs.subscribe(user => {
          console.log(`user name is ${user.name}`);
          console.log(`user email is ${user.email}`);
          console.log(`user website is ${user.website}`);
          console.log('-----------------------------');
},
error => console.error("error", error),
complete => console.log("completed"))
```

Manejo de llamadas asíncronas sobre observables

- map vs flatMap (mergemap)
  - la función asociada al map sobre un observable puede devolver un observable
  - es muy común que devuelva una promesa
  - en este caso se tienen observables anidados
  - es conveniente aplanar en un solo observable