# ReactiveX RxJS

## Reactive Extensions For JavaScript

Niccolò Maltoni

niccolo.maltoni@studio.unibo.it

# Introduzione

## Programmazione reattiva

*“ It is convenient to distinguish roughly between three kinds of computer programs. Transformational programs compute results from a given set of inputs; typical examples are compilers or numerical computation programs. Interactive programs interact at their own speed with users or with other programs; from a user point of view, a time-sharing system is interactive.* ***Reactive programs also maintain a continuous interaction with their environment, but at a speed which is determined by the environment, not the program itself****. Interactive programs work at their own pace and mostly deal with communication, while reactive programs only work in response to external demands and mostly deal with accurate interrupt handling. ”*

Gérard Berry, 1989 [1]

# Introduzione

## Reactive Manifesto [2]

- Prima versione presentata nel 2013
- Principi base:
  - **responsività**
  - **resilienza**
  - scalabilità
  - *event-driven*
- La versione attuale (2.0) è stata presentata l'anno successivo[1]
  - da scalabilità a **elasticità**
  - da *event-driven* a ***message-driven***

[1] `https://www.lightbend.com/blog/reactive-manifesto-20`

# Introduzione

## Reactive Manifesto

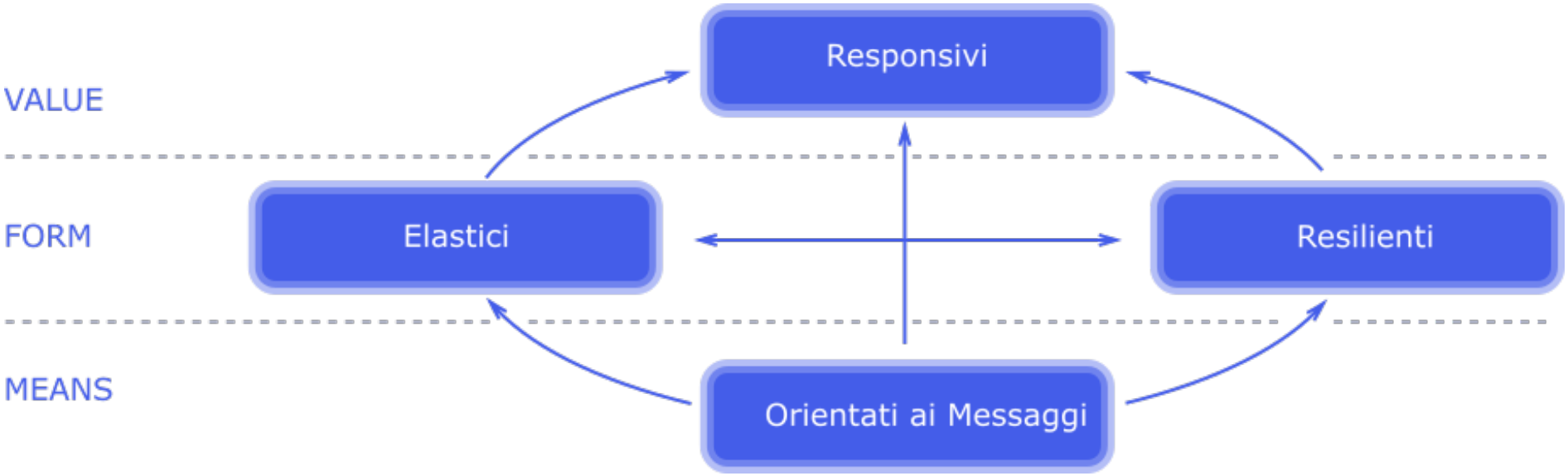



Figura: Rappresentazione dei principi base del *Reactive Manifesto*

## Responsività

- **tempestività** della risposta
  - rapida identificazione dei problemi
  - minimizzazione del tempo di risposta
- garanzia di **qualità del servizio** nel tempo
- **predicibilità** del comportamento
  - semplificazione della gestione degli errori
  - fiducia degli utenti finali nel sistema

## Resilienza

- il sistema resta **responsivo anche in caso di guasti**
- garantita tramite:
  - replica
  - contenimento
  - isolamento
  - delega

## Elasticità

- il sistema rimane **responsivo sotto carichi di lavoro variabili** nel tempo
- **adattabilità** attraverso incremento o decremento delle risorse allocate al processamento degli input
  - no sezioni contese né colli di bottiglia
  - distribuibilità
  - replica dei componenti
  - ripartizione degli input
- possibile implementazione predittiva

## *Message-Driven*

- **scambio di messaggi asincrono**
  - stile di comunicazione non bloccante
- basso accoppiamento tra i componenti
- isolamento e trasparenza sul dislocamento
- esprimere i guasti del componente sotto forma di messaggi

# Introduzione

## Reactive Extensions

- Reactive Extensions (o **ReactiveX**) sono un set di librerie per la maggior parte dei linguaggi e dei framework moderni che permette l'implementazione di programmi in grado di operare su sequenze di dati e input in modo *reattivo* e *asincrono*, indipendentemente dalla natura della sorgente
- ispirate alle *Microsoft's Reactive Extensions* per ambiente .NET
- combinazione di:
  - pattern Observer
  - pattern Iterator
  - programmazione funzionale

# RxJS: Reactive Extensions Library for JavaScript

## Reactive Extensions per JavaScript (e TypeScript)[2]

- RxJS è una libreria per la programmazione reattiva pensata per JavaScript
- semplifica l'implementazione di *callback* e chiamate asincrone attraverso il tipo `Observable` e i suoi costrutti satelliti, come:
  - `Observer`
  - `Schedulers`
  - `Subjects`
  - operatori come `map`, `filter`, `reduce`, `every`, ecc ...

[2] `http://reactivex.io/rxjs`

# RxJS: Reactive Extensions Library for JavaScript

## Installazione

- Via **npm**:

```
npm install @reactivex/rxjs
```

- Via **CDN**:

```
<script src="https://unpkg.com/rxjs/bundles/rxjs.umd.min.js"></script>
```

# Costrutti fondamentali

## `Observable`

- costituisce il costrutto base della libreria
- è uno *stream*, costruito in modo asincrono col procedere del tempo
- è in grado di generare valori utilizzando funzioni pure
- offre operatori per il controllo di flusso
- può essere costruito:
  - a partire da eventi (ad esempio del DOM), con `fromEvent()`
  - a partire da `promise`
  - da zero, con `timer`
  - da zero, con `interval`
  - da qualsiasi cosa, con `of`

# Costrutti fondamentali

## Observable: Hot VS Cold

### Cold

Uno *stream* si definisce ***cold*** se l'`Observable` genera il produttore dei valori nel flusso

```
const source = from([1, 2, 3, 4, 5]);
source.subscribe(val => console.log(val));
```

### Hot

Uno *stream* si definisce ***hot*** se l'`Observable` racchiude il produttore dei valori nel flusso

```
const source = fromEvent(document, 'click')
    .pipe(map(event => event.timeStamp));
source.subscribe(val => console.log(val));
```

## `.pipe()`

- A partire da RxJS 5.5, sono stati introdotti i *pipable operators*
- da RxJS 6, essi sono gli operatori standard, trovabili in `rxjs/operators`
- l'operatore `.pipe()` prende un infinito numero di argomenti, che sono funzioni applicabili allo stream; ad esempio:

```
const stream = interval(350).pipe(
    take(25),
    map(gaussian),
    map(num => "•".repeat(Math.floor(num * 65)))
);
stream.subscribe(dot => (console.log(dot)));
```

# Costrutti fondamentali

## Observable: Pipable operators

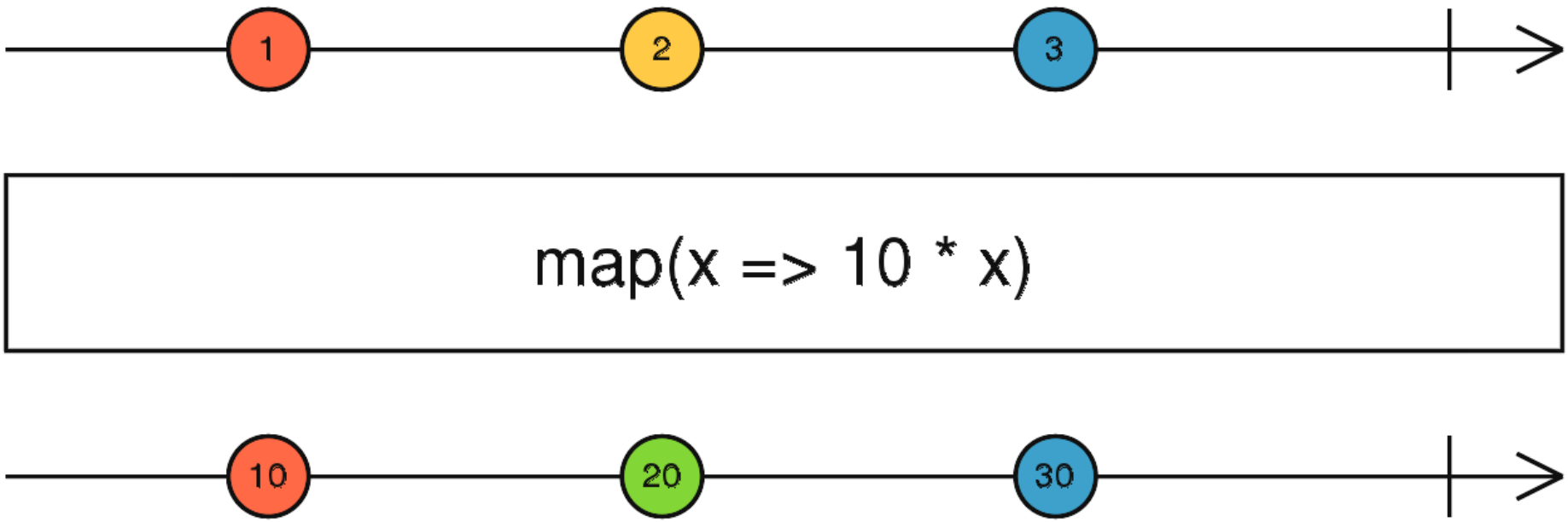
### `.map()`

- simile alla funzione `.map()` degli array
- applica una **funzione proiezione** ad ogni valore emesso dall'`Observable` sorgente ed emette i valori risultanti come `Observable`
- utile per il parsing di JSON

# Costrutti fondamentali

## Observable: Pipable operators: `.map()`

```
const source = from([1, 2, 3])
    .pipe(
        map(x => 10 * x)
    );
source.subscribe(val => console.log(val));
```



# Costrutti fondamentali

## Observable: Pipable operators

### `.reduce()`

- simile alla funzione `.reduce()` degli array
- applica una **funzione accumulatore** sull'`Observable` sorgente e ritorna il risultato una volta che la sorgente termina

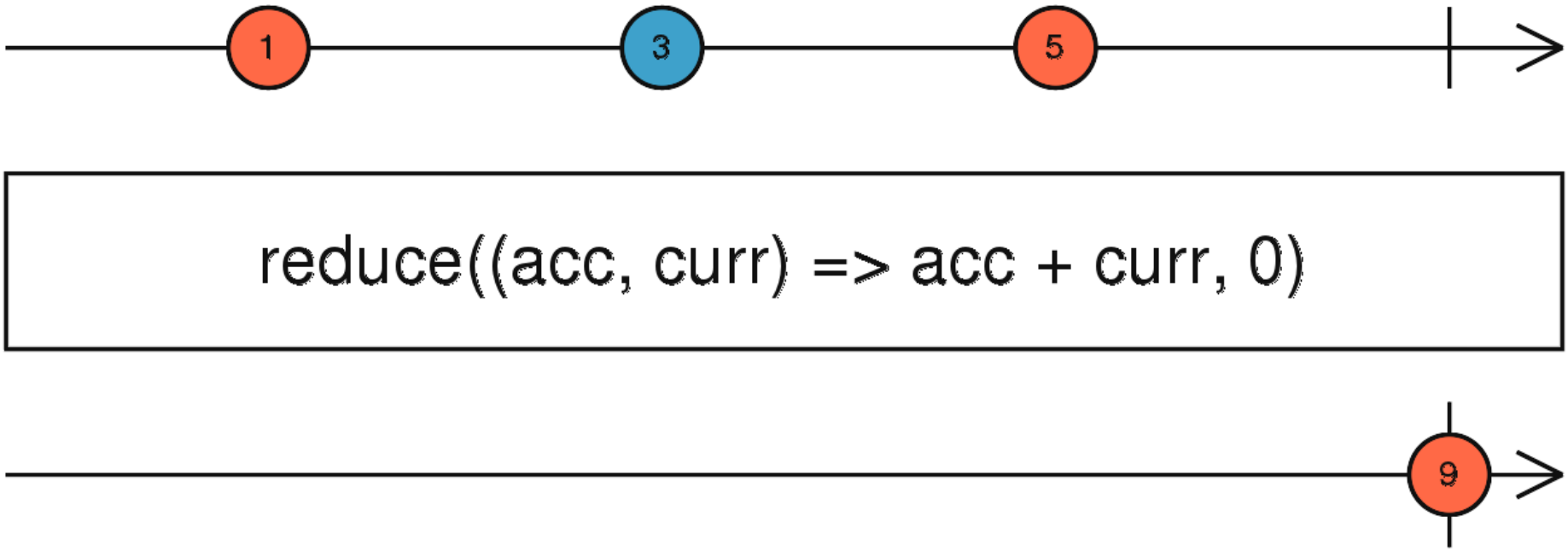
### `.scan()`

- applica una **funzione accumulatore** sull'`Observable` sorgente e ritorna ogni risultato intermedio
- simile alla `.reduce()`, ma emette un valore ad ogni valore emesso dalla sorgente anziché attenderne il completamento

# Costrutti fondamentali

## Observable: Pipable operators: `.reduce()` & `.scan()`

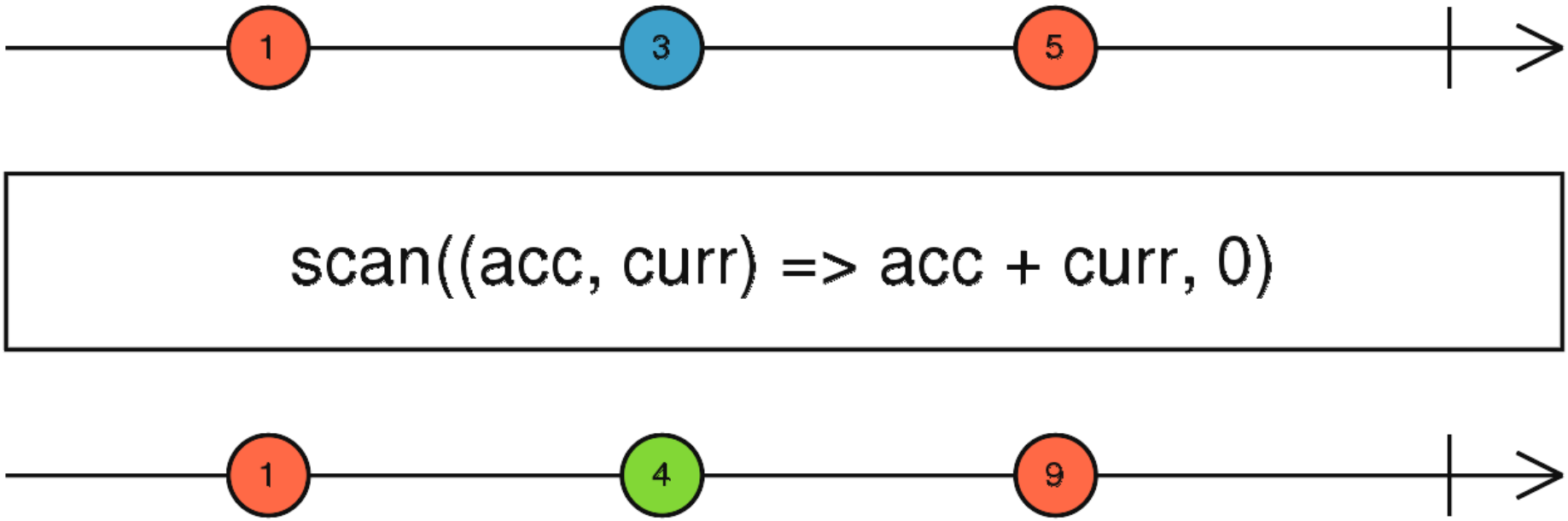
```
const source = from([1, 2, 3])
    .pipe(
        reduce((acc, curr) => acc + curr, 0)
    );
source.subscribe(val => console.log(val));
```



# Costrutti fondamentali

## Observable: Pipable operators: `.reduce()` & `.scan()`

```
const source = from([1, 2, 3])
    .pipe(
        scan((acc, curr) => acc + curr, 0)
    );
source.subscribe(val => console.log(val));
```



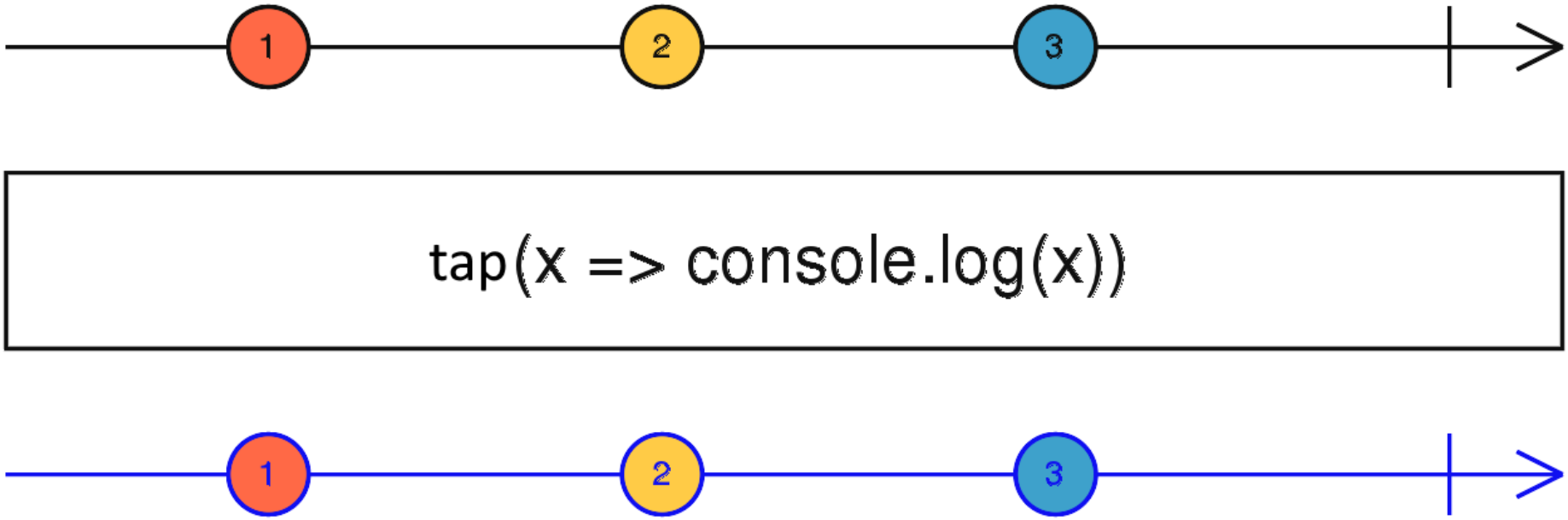
## `.tap()` (precedentemente `.do()`)

- la versione "*pipable*" è stata rinominata da `.do()` a `.tap()` per conflitto di nomi con funzioni standard di JavaScript
- esegue un *side effect* per ogni valore emesso dall'`Observable` sorgente, ma lo ritorna invariato
- estremamente utile in fase di debug

# Costrutti fondamentali

## Observable: Pipable operators: `.tap()`

```
const source = from([1, 2, 3])
    .pipe(
        tap(val => console.log(val))
    );
```

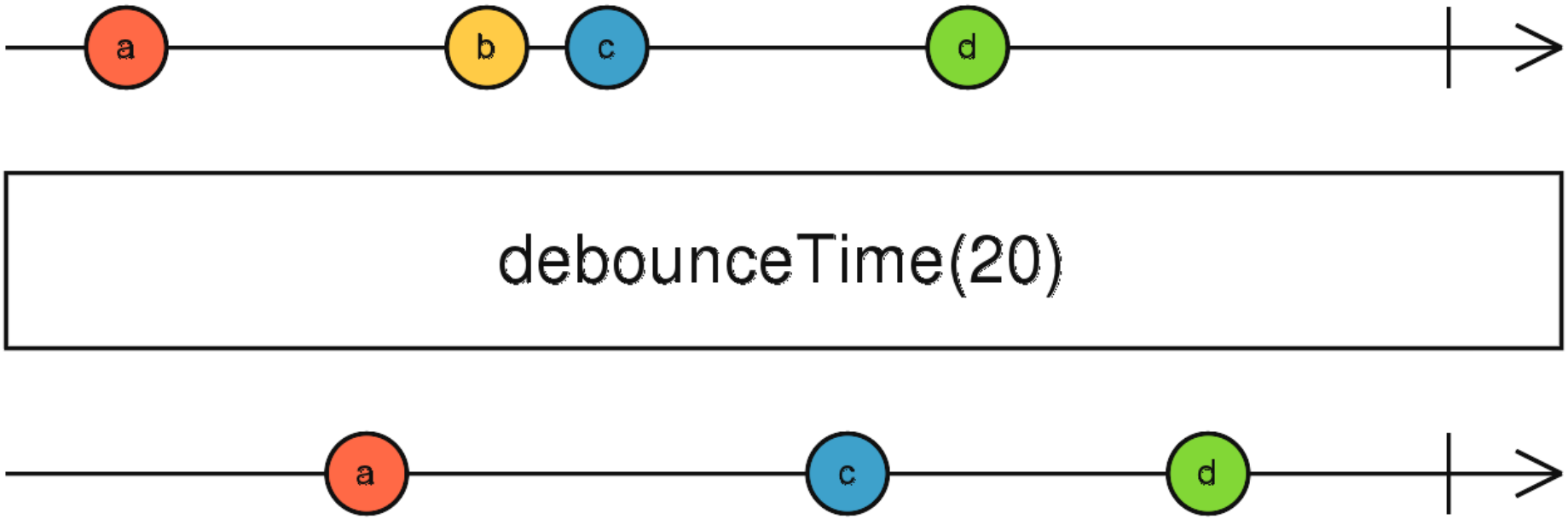


## `.debounce()` & `.debounceTime()`

- Permettono di filtrare gli eventi emessi dall'`Observable` sorgente, emettendo un valore solo dopo che è passato un certo numero di millisecondi da quando non sono stati emessi valori
- `.debounceTime()` permette di specificare il lasso di tempo entro il quale effettuare il *debounce*
- `.debounce()` si comporta come `.debounceTime()`, ma il periodo di silenzio è determinato da un altro `Observable`
- utili per attendere che l'utente abbia terminato di fare qualcosa con un componente osservato prima di gestire l'evento

# Costrutti fondamentali

## Observable: Pipable operators: `.debounceTime()`

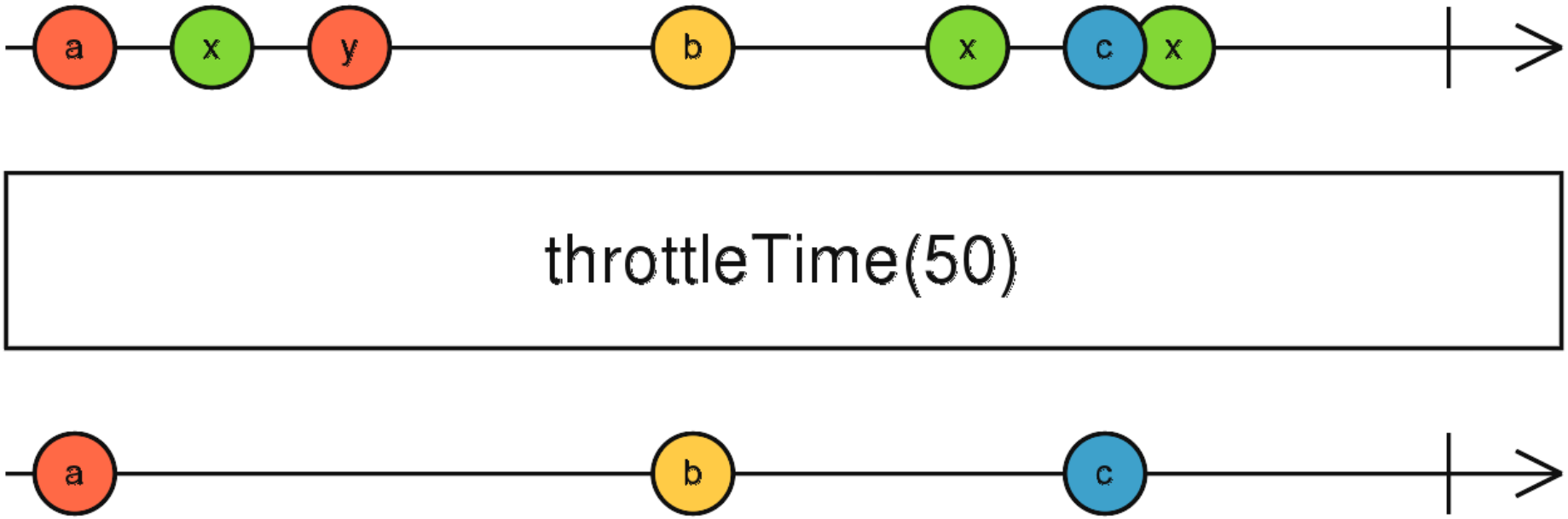


## `.throttle()` & `.throttleTime()`

- Permettono di filtrare gli eventi emessi dall'`Observable` sorgente, emettendo un valore solo ogni un certo lasso di tempo
- `.throttleTime()` emette un valore dall'`Observable` sorgente ed ignora ulteriori valori per i millisecondi specificati
- `.throttle()` si comporta come `.throttleTime()`, ma il periodo di silenzio è determinato da un altro `Observable`
- utili per ridurre il numero di input a livelli accettabili

# Costrutti fondamentali

## Observable: Pipable operators: `.throttleTime()`



## `.catchError()` (precedentemente `.catch()`)

- la versione "*pipable*" è stata rinominata da `.catch()` a `.catchError()` per conflitto di nomi con funzioni standard di JavaScript
- permette di catturare gli errori nel flusso dell'`Observable` e gestirli attraverso un *handler*; fatto ciò, può:
  - ritornare un nuovo `Observable` con l'errore gestito
  - lanciare un errore

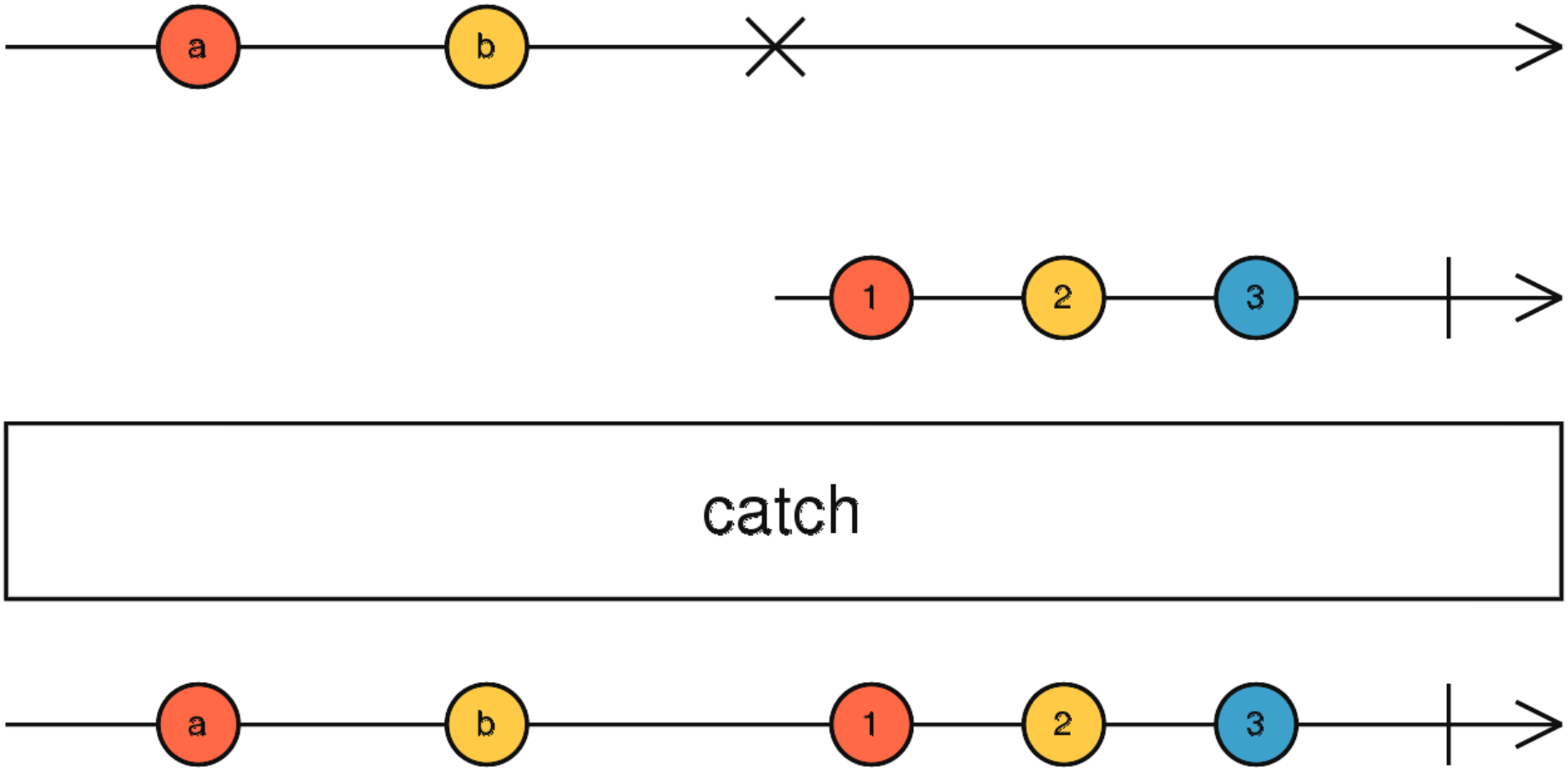
# Costrutti fondamentali

## Observable: Pipable operators: `.catchError()`

```
Observable.of('a', 'b', 'c', 'd', 'e')
    .pipe(
        map(c => { if (c == 'c') { throw 'c!'; } return c; }),
        catch(err => Observable.of(1, 2, 3)),
    )
    .subscribe(x => console.log(x));
```

# Costrutti fondamentali

## Observable: Pipable operators: `.catchError()`



# Costrutti fondamentali

## `Subject`

- Un `Subject` è sia un `Observable` che un `Observer`:
  - è possibile sottoscrivere tutti gli `Observer` che servono a un `Subject`, dopodiché sottoscrivere il `Subject` a una sorgente di *backend*
  - è possibile implementare un `Observable` con *caching*, *buffering* e/o *time-shifting*
  - è possibile utilizzare un `Subject` per fare *broadcast* di eventi a più `Subscriber`
  - è possibile aggiungere valori ad un `Subject` che racchiude un `Observable`, tramite l'operatore `.next()`

## Conclusioni

- La programmazione reattiva sta diventando un elemento molto importante della programmazione moderna, non solo *web-oriented*
- Il set di librerie fornito da ReactiveX è una delle implementazioni più famose per i vari linguaggi, ma ne esistono molte altre, ad esempio:
  - Bacon.js
  - Kefir.js
  - meccanismi di programmazione asincrona combinati a librerie per la manipolazione di stream
  - ecc ...
- Angular sfrutta RxJS internamente e la sta mettendo in evidenza presso la community come punto di riferimento

# Riferimenti bibliografici I



G. Berry. *Real time programming: special purpose or general purpose languages*. Research Report RR-1065. INRIA, 1989. URL: https://hal.inria.fr/inria-00075494.


J. Bonér, D. Farley, R. Kuhn e M. Thompson. «The Reactive Manifesto». In: (set. 2014). URL: http://www.reactivemanifesto.org/.


ReactiveX. *Reactive Extension main website*. URL: http://reactivex.io.


ReactiveX. *RxJS GitHub repository*. URL: https://github.com/ReactiveX/rxjs.


A. Ricci. *module-2.4 - Reactive Programming*. URL: http://campus.unibo.it/333427/.