PARADIGMI DI PROGRAMMAZIONE

A.A. 2021/2022

Laurea triennale in Informatica

20: Reattività

REACTIVE EXTENSIONS

Per cercare astrazioni di livello più altro con cui gestire, in modo coordinato, le problematiche di sistemi concorrenti e distribuiti facciamo un passo indietro, recuperando uno strumento di cui abbiamo già parlato, rileggendolo sotto un'altra luce.

L'astrazione di esecuzione fornita dallo Stream si basa sull'inversione del controllo dell'iterazione.

E' lo Stream che avanza l'esecuzione, e che decide la sua struttura.

Tuttavia, alcune esigenze rimangono aperte:

- manca un protocollo esplicito per gestire la terminazione dello stream
- gli errori sono gestiti come eccezioni

Nel 2009, il gruppo di Erik Mejer introduce in .NET 4.0 la versione 1.0 delle Reactive Extensions.

"ReactiveX is a library for composing asynchronous and event-based programs by using observable sequences."

Le Reactive Extension forniscono una semantica per la definizione di elaborazioni asincrone di sequenze di oggetti.

E' molto di più di una API: è un intero modello di esecuzione.

Il lavoro teorico descrive così precisamente il modello che viene subito avviato il *porting* verso altre piattaforme.

Netflix rilascia RxJava nel 2014.

Le basi del modello sono i seguenti concetti:

- Observable
- Scheduler
- Subscriber
- Subject

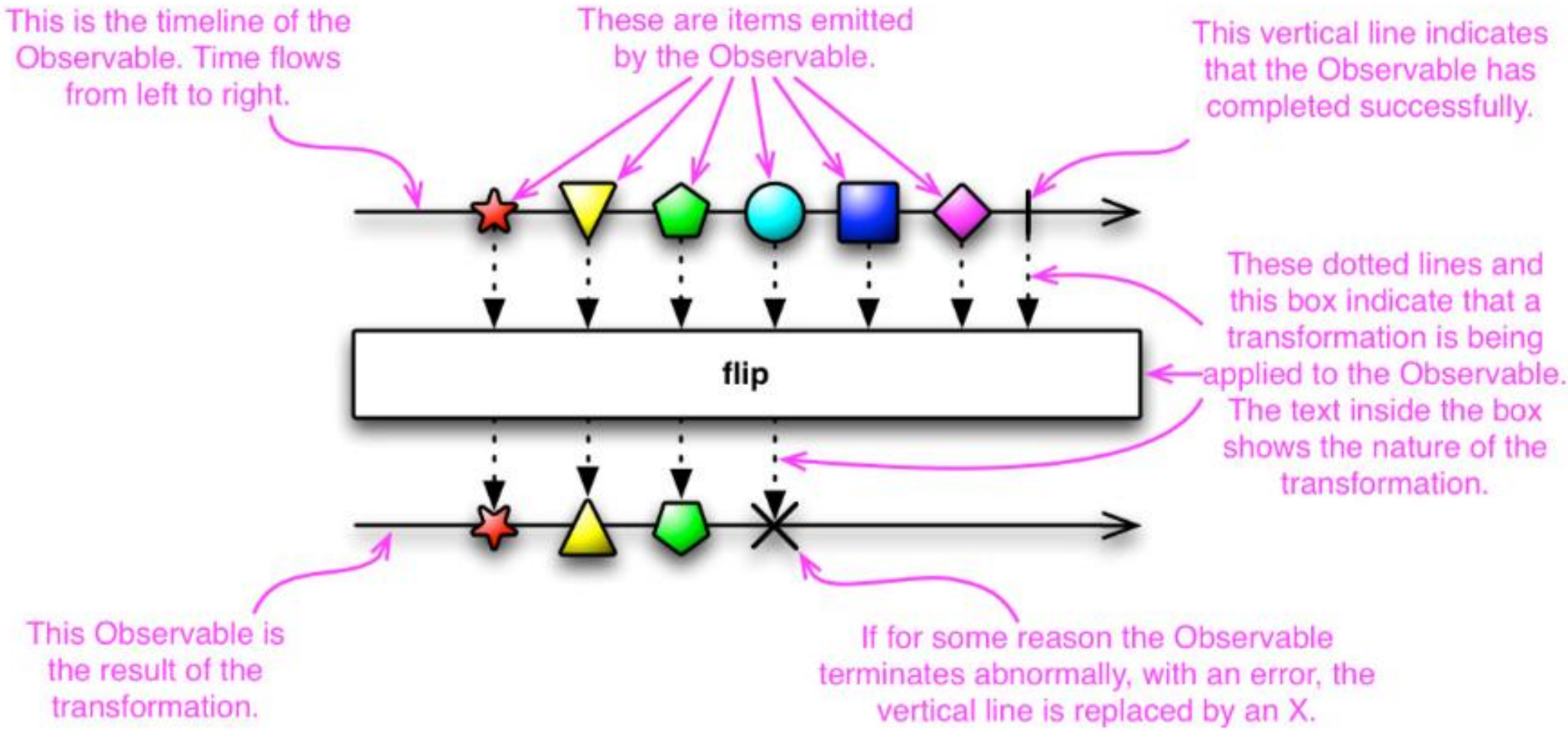
L'idea principale del modello è una implementazione dell'**Observer pattern** *done right*.

Un `Observable` in Rx è un oggetto concettualmente simile ad uno stream, che emette *nel tempo* una sequenza di valori.

E' possibile trasformare, filtrare ed elaborare questi valori in modo esteriormente simile alle analoghe implementazioni su di uno stream.

E' possibile *osservare* i valori emessi da un `Observable` fornendo il comportamento da adottare in caso di:

- valore ricevuto
- eccezione lanciata da un precedente componente
- termine del flusso di dati


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.

evento	Iterable	Observable
successivo	`T next()`	`onNext(T)`
errore	lancia `Exception`	`onError(E)`
completamento	`!hasNext()`	`onCompleted()`

it.unipd.pdp2021.rx.RxPerfect

```
System.out.println("Defining...");
Observable.range(0, 10000).map(new RxDivisors())
  .filter(new RxPerfectPredicate())
  .subscribe((c) -> {
    System.out.println(c);
  }, (t) -> {
    t.printStackTrace();
  }, () -> {
    System.out.println("Done");
  });
System.out.println("Defined");
```

Il risultato è:

- una semantica più ricca
- maggiore regolarità nella composizione
- indipendenza dal modello di esecuzione (sincrono/asincrono)

La maggior parte degli operatori sugli `Observable` accettano uno `Scheduler` come parametro.

Ogni operatore può così essere reso concorrente; lo `Scheduler` scelto permette di indicare il tipo di concorrenza desiderato.

it.unipd.pdp2021.rx.Scheduler

```
System.out.println("Defining...");
Observable.range(0, 1000000).map(new RxDivisors())
  .filter(new RxPerfectPredicate())
  .subscribeOn(Schedulers.computation())
  .subscribe(
    (c) -> { System.out.println(c); },
    (t) -> { t.printStackTrace(); },
    () -> { System.out.println("Done"); done[0] = true; });
System.out.println("Defined");
while (!done[0]) Thread.sleep(1000);
System.out.println("End");
```

Un `Subscriber` rappresenta un ascoltatore di un `Observable`: fornisce il codice che reagisce agli eventi per ottenere il risultato finale dalla catena di elaborazione.

Un `Subject` può consumare uno o più `Observable`, per poi comportarsi esso stesso da `Observable` e quindi introdurre modifiche sostanziali nel flusso degli eventi.

Lo schema concettuale proposto da Rx è estremamente utile per:

- costruire stream di elaborazione complessi e asincroni
- fornire un'interfaccia semplice ma facilmente componibile per trattare successioni di eventi nel tempo

- scrivere algoritmi facili da portare da un linguaggio all'altro
- strutturare una elaborazione concorrente di uno stream di dati su garanzie solide

- gestire gli eventi provenienti da una UI con la stessa semplicità di dati provenienti da un file, da una sequenza di dati, o altra sorgente.

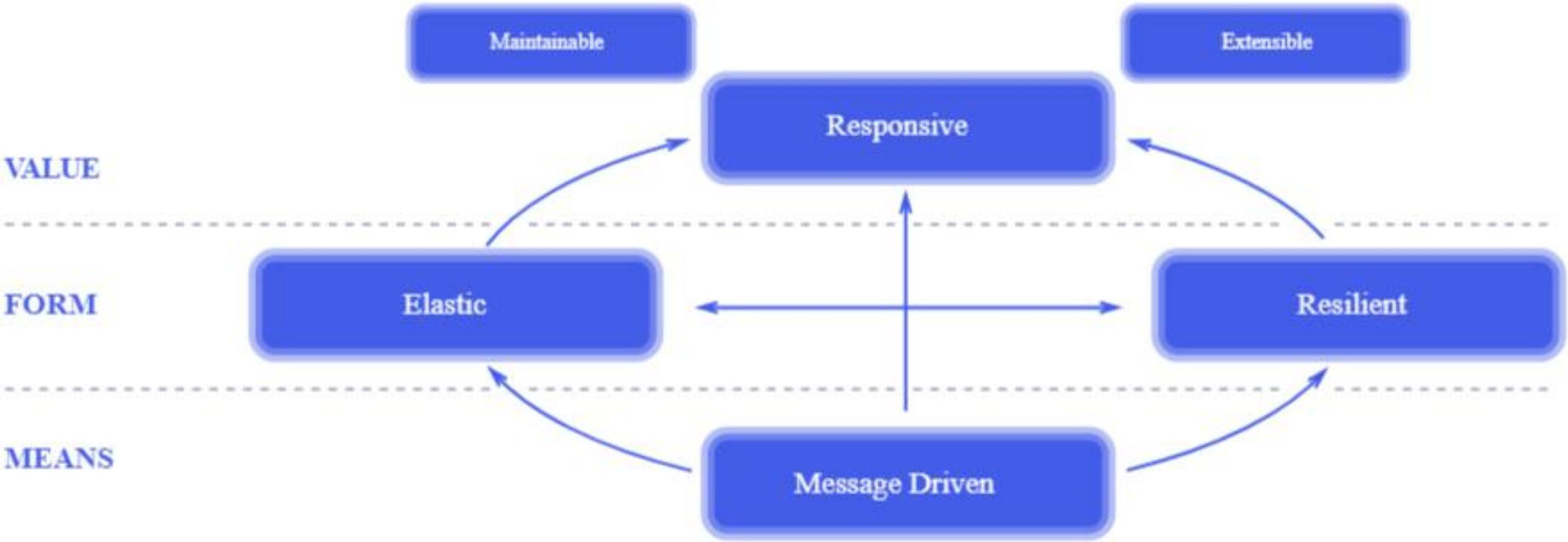
REACTIVE MANIFESTO

Le Reactive Extensions avviano presto un nuovo filone di ricerca per studiare come il modello da loro proposto si adatta alle necessità emergenti delle applicazioni di elaborazione di stream di dati, che nella prima metà degli anni 2010 diventano sono ormai comuni.

Per dare una definizione precisa a questo nuovo tipo di applicazioni, e stabilire un concetto che potesse catalizzare il discorso della ricerca in una direzione ben precisa, alcuni autori di tecnologie del campo definiscono e pubblicano il Reactive Manifesto.

Il *Reactive Manifesto* definisce le caratteristiche dei sistemi *reattivi*:

- Pronti alla risposta (*Responsive*)
- Resilienti (*Resilient*)
- Elastici (*Elastic*)
- Orientati ai messaggi (*Message Driven*)


Maintainable
Extensible
Responsive
VALUE
FORM
Elastic
Resilient
MEANS
Message Driven

Un sistema Reattivo deve essere *Pronto alla risposta* (Responsive) in quanto la bassa latenza di interazione è un principio cardine dell'usabilità.

Un sistema quindi deve privilegiare la possibilità fornire una risposta sempre ed in un tempo prevedibile e costante. Un errore è una risposta come le altre; ed allo stesso modo deve essere individuato e comunicato con le stesse tempistiche.

Un sistema Reattivo deve essere *Resiliente* (*Resilient*), ovvero gestire i fallimenti continando a rispondere con la stessa prontezza.

La resistenza agli errori si ottiene con la replicazione di componenti isolati, la coscienza che ogni parte del sistema può fallire, e la rapida creazione di nuovi componenti in sostituzione di quelli che sono andati in errore o non sono più disponibili.

Un sistema Reattivo deve essere *Elastico* (*Elastic*), cioè in grado di consumare una quantità variabile di risorse in funzione del carico in ingresso.

Il mantenimento della latenza di risposta prevista si ottiene distribuendo il carico su di un maggior numero di risorse nel modo più lineare possibile, senza colli di bottiglia o punti di conflitto, suddividendo gli input in *shard* distribuite automaticamente. L'obiettivo è una scalabilità efficace ed economica su hardware non specializzato.

Un sistema Reattivo deve essere *Orientato ai messaggi* (*Message Driven*) perché questa primitiva di comunicazione abilita le altre caratteristiche.

Attraverso lo scambio di messaggi i componenti possono rimanere disaccoppiati, si possono indirizzare anche su nodi distribuiti, ed il carico può essere suddiviso fra copie in esecuzione su nodi differenti. L'asincronia della comunicazione e l'assenza di blocchi permettono di consumare al meglio le risorse disponibili.

Queste caratteristiche impongono una organizzazione architetturale ben precisa; il *Manifesto* intende dirigere lo sviluppo tecnologico in una precisa direzione vista come quella più adatta a supportare sistemi con le caratteristiche desiderate.

L' *Orientamento ai messaggi* è il mezzo con cui il sistema si struttura in forma *Elastica* e *Resiliente* per ottenere il valore della *Prontezza alla risposta*.

Sottoprodotti di questa architettura sono componenti manutenibili e facili da estendere, per inseguire il rapido cambiamento dei requisiti.

BIG VS FAST

Negli stessi anni in cui aumentava l'interesse per ReactiveX come piattaforma, il problema del *Big Data* diventava sempre più importante e mutava in qualcosa più difficile da trattare.

Qualche autore comincia a chiamare il problema **Fast Data**: dati di dimensioni paragonabili al Big Data, in arrivo continuo; impensabile e/o inutile persisterli per elaborarli a partire dal supporto di salvataggio.

Per il loro volume, per la velocità di arrivo, e la bassa latenza richiesta nel reagire alle informazioni estratte, è necessario trattarli in diretta via via che si presentano.

Nel costruire un sistema massicciamente parallelo per estrarre informazioni da un flusso di dati continuo, uno dei principali problemi è l'armonizzazione delle varie differenze di velocità di elaborazione fra i vari componenti.

La soluzione fino ad allora impiegata, cioè l'esecuzione di *batch* massivamente paralleli, non poteva più funzionare in quanto introduceva latenze superiori alla periodo di utilità delle informazioni estratte.

Il componente più lento diventerà il collo di bottiglia e stabilirà la velocità massima dell'elaborazione, per poi fallire soverchiato dai dati che arrivano troppo velocemente.

Da questo problema nasce la necessità di aggiungere alla semantica delle Reactive Extension, che appaiono particolarmente adatte a questo tipo di problema, un nuovo concetto: la **back-pressure**.

Con **back-pressure** si intende la resistenza che il componente successivo può opporre ai dati provenienti dal componente precedente della catena di elaborazione.

Questo concetto permette ad ogni nodo della catena di dichiarare quanti dati è in grado di gestire.

REACTIVE STREAMS

"The main goal of Reactive Streams is to govern the exchange of stream data across an asynchronous boundary while ensuring that the receiving side is not forced to buffer arbitrary amounts of data."

http://www.reactive-streams.org/

Reactive Streams aumenta le interfacce e le garanzie fornite da ReactiveX introducendo l'esplicita gestione della *back-pressure* per impedire che un nodo possa essere soverchiato dai dati inviati dal nodo precedente, o sovraccaricare quello successivo.

Inoltre, viene inclusa nella considerazione dello standard anche la casistica in cui i diversi componenti di uno Stream non si trovino nello stesso nodo, ma siano distribuiti.
Sono comprese sia le API asincrone, sia i protocolli di comunicazione.

Anche Reactive Streams parte da una modello semantico; oltre a questo, ha a disposizione un Technology Compatibility Kit per verificare una implementazione candidata.

Mentre ReactiveX è poliglotta, Reactive Streams è multi-implementazione, ma concentrato su JVM e Javascript.

In questo modo, la mission di Reactive Streams comprende non solo le interfacce applicative, ma anche i protocolli di rete: la *back-pressure* infatti è necessaria non solo fra thread, ma anche fra nodi di calcolo distribuito.

Esistono molteplici implementazioni che soddisfano il TCK:

- Akka Streams
- MongoDB java Driver
- RxJava
- Vertx Reactive Streams
- Java 9 java.util.concurrent.Flow

Il modello concettuale di Reactive Streams è compatto tanto quanto quello di ReactiveX:

- Publisher
- Subscriber
- Subscription
- Processor

Un `Publisher` fornisce un numero potenzialmente infinito di elementi in sequenza, rispettando le richieste dei suoi `Subscriber`.

```
public interface Publisher< T > {
  public void subscribe(Subscriber< ? super T > s);
}
```

Un `Subscriber` consuma gli elementi forniti da un `Publisher` ed è in grado di controllare il flusso degli elementi in arrivo.

```
public interface Subscriber< T > {
  public void onSubscribe(Subscription s);
  public void onNext(T t);
  public void onError(Throwable t);
  public void onComplete();
}
```

Una `Subscription` rappresenta il legame fra un `Subscriber` ed un `Publisher` e permette di controllarlo, per esempio interrompendolo.

```
public interface Subscription {
  public void request(long n);
  public void cancel();
}
```

Un `Processor` è sia un `Subscriber` che un `Publisher`, e deve sottostare ad entrambi i contratti. Rappresenta uno snodo di elaborazione intermedio in grado di alterare il flusso di elementi aggregando più `Publisher` o controllando più `Subscriber`.

```
public interface Processor< T, R >
  extends Subscriber< T >, Publisher< R > {
}
```

L'API dei Reactive Streams è stata inclusa, a partire da Java 9, nella libreria standard, usando come base la classe `java.util.concurrent.Flow`, con lo scopo di promuoverne la standardizzazione.

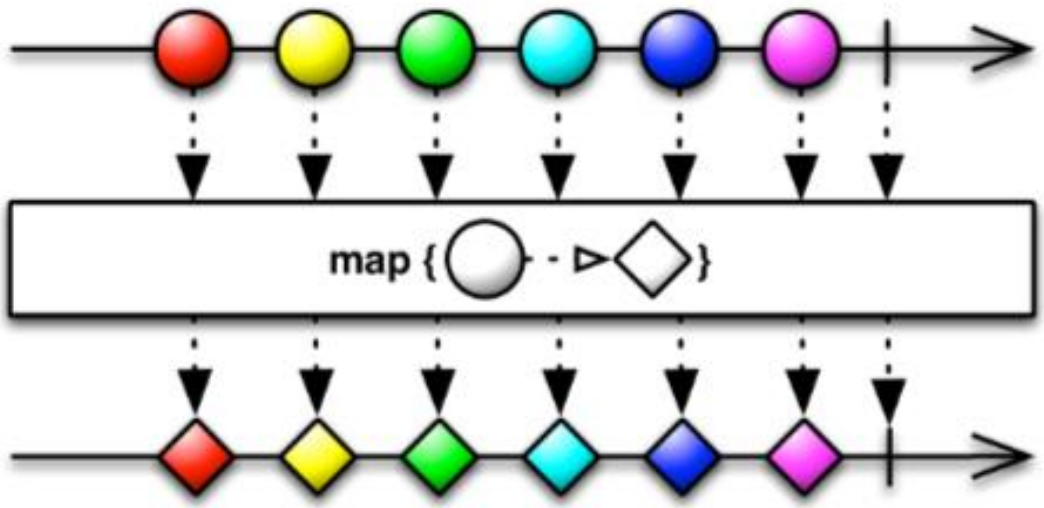
OPERATORI

Vediamo alcuni esempi di operatori per avere un'idea di come si può lavorare con un Reactive Stream.

Faremo riferimento alla documentazion delle Reactive Extensions perché gli operatori di base sono i medesimi; a seconda della classe usata (`Observable` oppure `Publisher`) si comportano in modo appropriato.

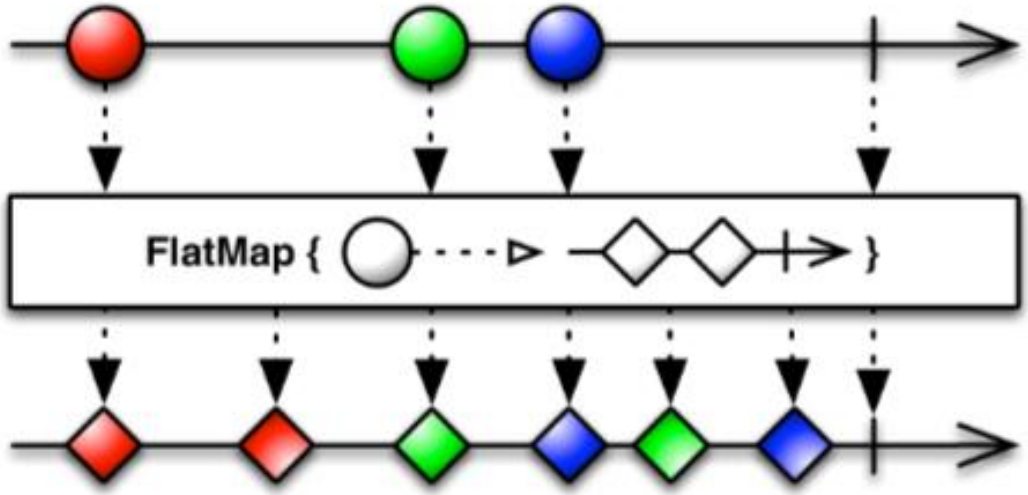
MAP

Trasforma gli elementi di uno stream, ottenendo uno stream di elementi trasformati.



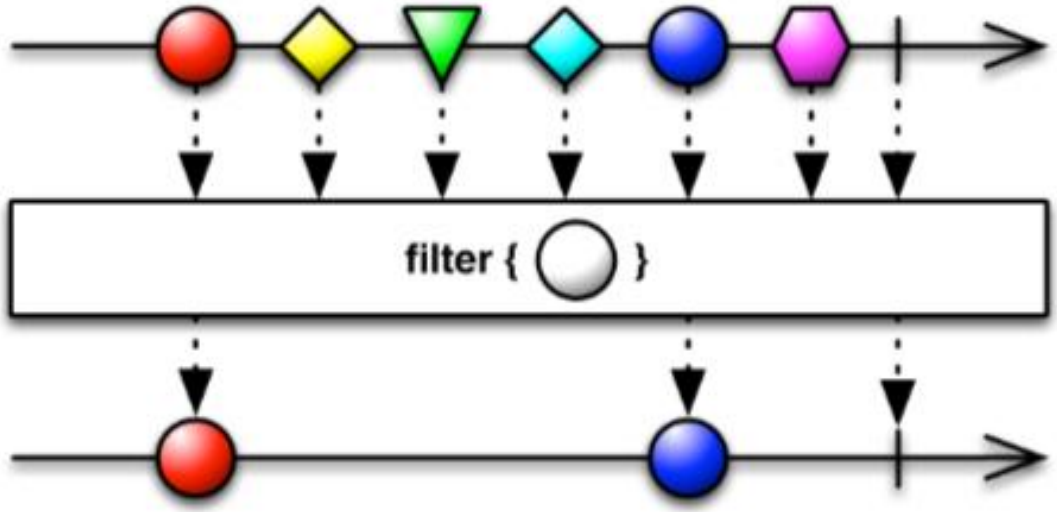
FLATMAP

Trasforma gli elementi di uno stream, concatenando i risultati in un solo stream.



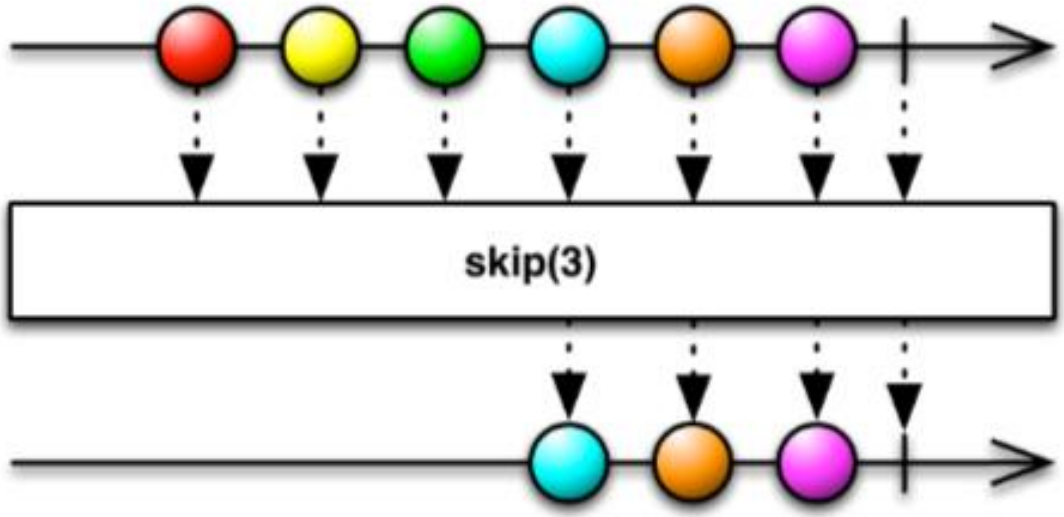
FILTER

Emette uno stream contenente solo gli elementi che soddisfano un predicato.



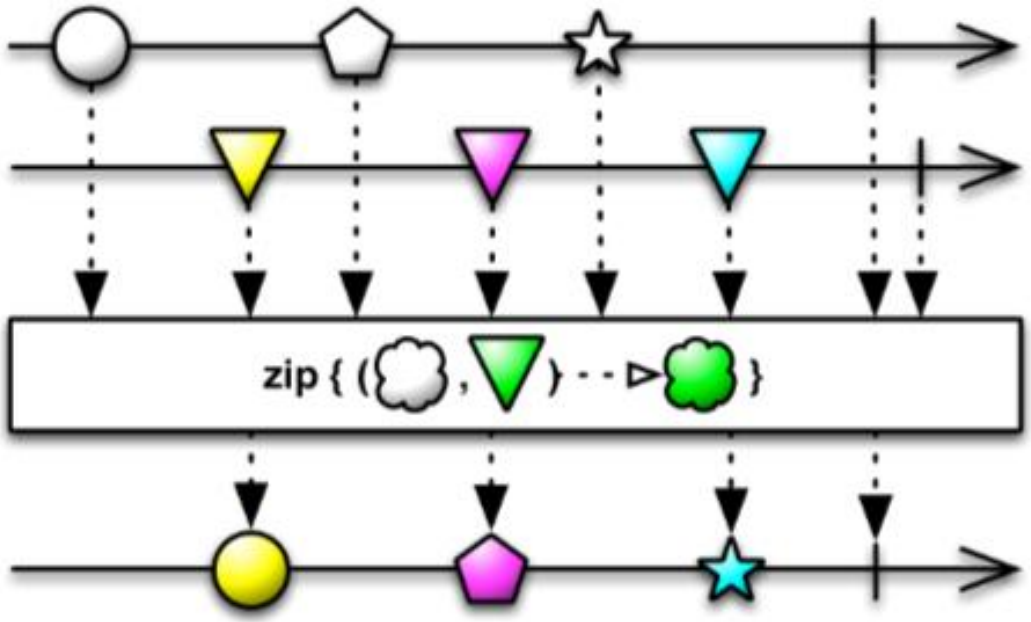
SKIP

Emette uno stream saltando i primi N elementi della sorgente.



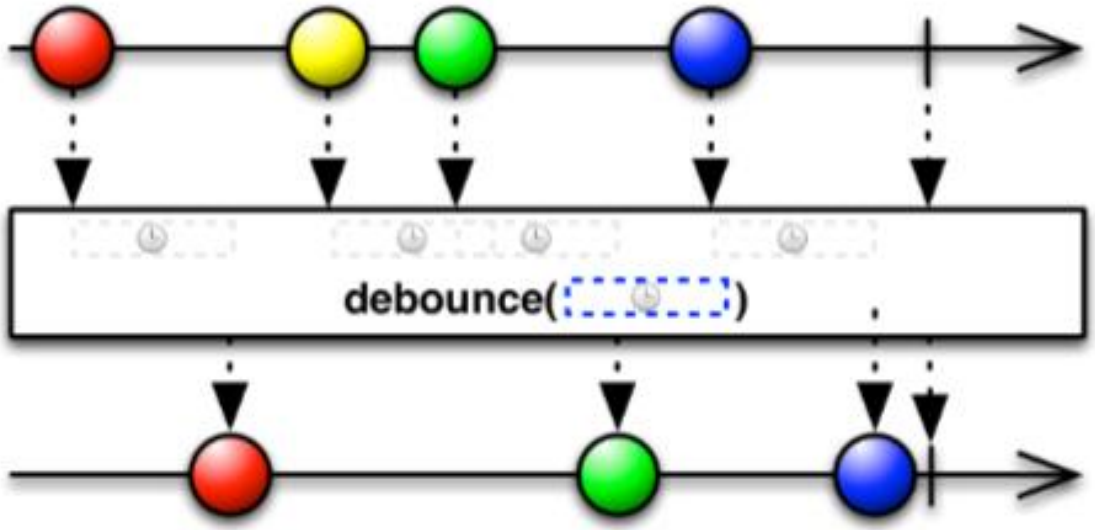
ZIP

Emette uno stream combinando a coppie elementi di due stream in ingresso.



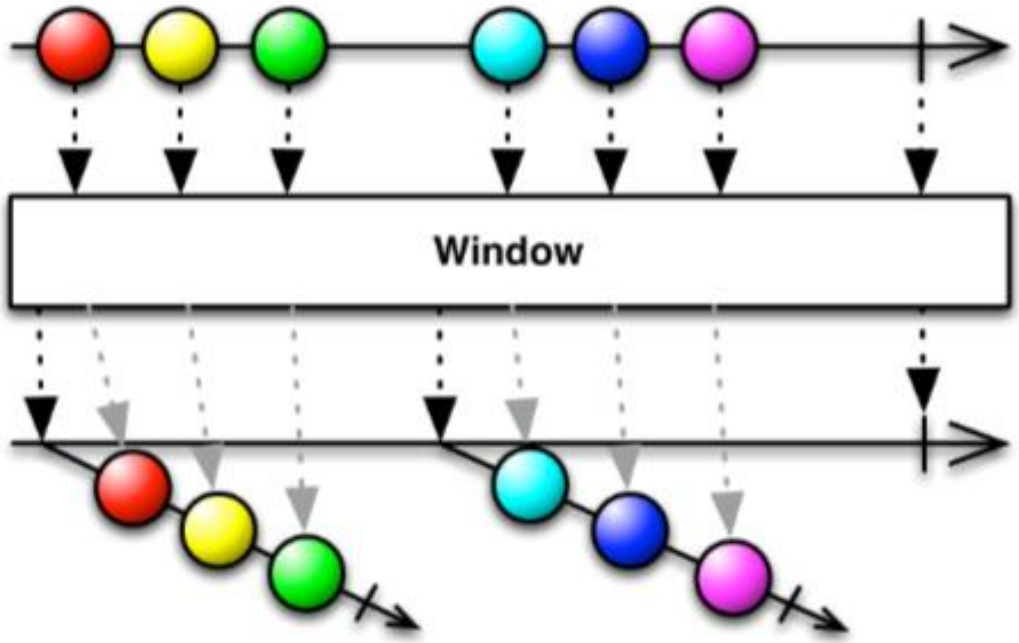
DEBOUNCE

Emette un elemento solo se è passato un lasso di tempo dall'ultimo elemento della sorgente.



WINDOW

Emette uno stream di partizioni dello stream sorgente.



PARALLELISMO

L'asincronia nell'esecuzione dei vari operatori è definita dallo Schedulatore usato per osservare un `Observable` o definire un operatore di uno stream.

Ciascuna implementazione fornisce degli schedulatori in relazione alla piattaforma in cui opera. Come abbiamo visto, `RxJava` permette di ottenerli a partire da metodi statici dell'oggetto `Schedulers`

Metodo	Schedulatore
.io()	Per stream legati alle operazioni di IO
.single()	Usa un singolo thread
.computation()	Per operatori legati al calcolo
.from(ex)	Usa l'`Executor` fornito

In `RxJava`, l'operatore `parallel()` permette di indicare che uno stream, da un certo punto in poi, va costruito come parallelo.

In questa modalità, solo alcuni operatori sono consentiti (per questioni di semantica) ed è necessario specificare lo schedulatore da usare con il metodo `.runOn(scheduler)`.

Il metodo `sequential()` indica che da quel punto in poi la pipeline di elaborazione va nuovamente intesa come sequenziale.

A differenza degli `Stream` della libreria di base, è possibile indicare una precisa sezione della pipeline che viene configurata parallelamente.

it.unipd.pdp2021.rx.Parallel

```
System.out.println("Defining...");
Flowable.range(0, 1000000).parallel(4)
  .runOn(Schedulers.computation()).map(new RxDivisors())
  .filter(new RxPerfectPredicate())
  .sequential().subscribe((c) -> {
      System.out.println(c);
    }, (t) -> {
      t.printStackTrace();
    }, () -> {
      System.out.println("Done");
      done[0] = true;
    });
System.out.println("Defined");
while (!done[0]) Thread.sleep(1000);
```