From Streams to Reactive Streams

Oleg Tsal-Tsalko

JUG Lviv 2017

ABOUT ME

OLEG TSAL-TSALKO

SOLUTION ARCHITECT AT EPAM SYSTEMS.

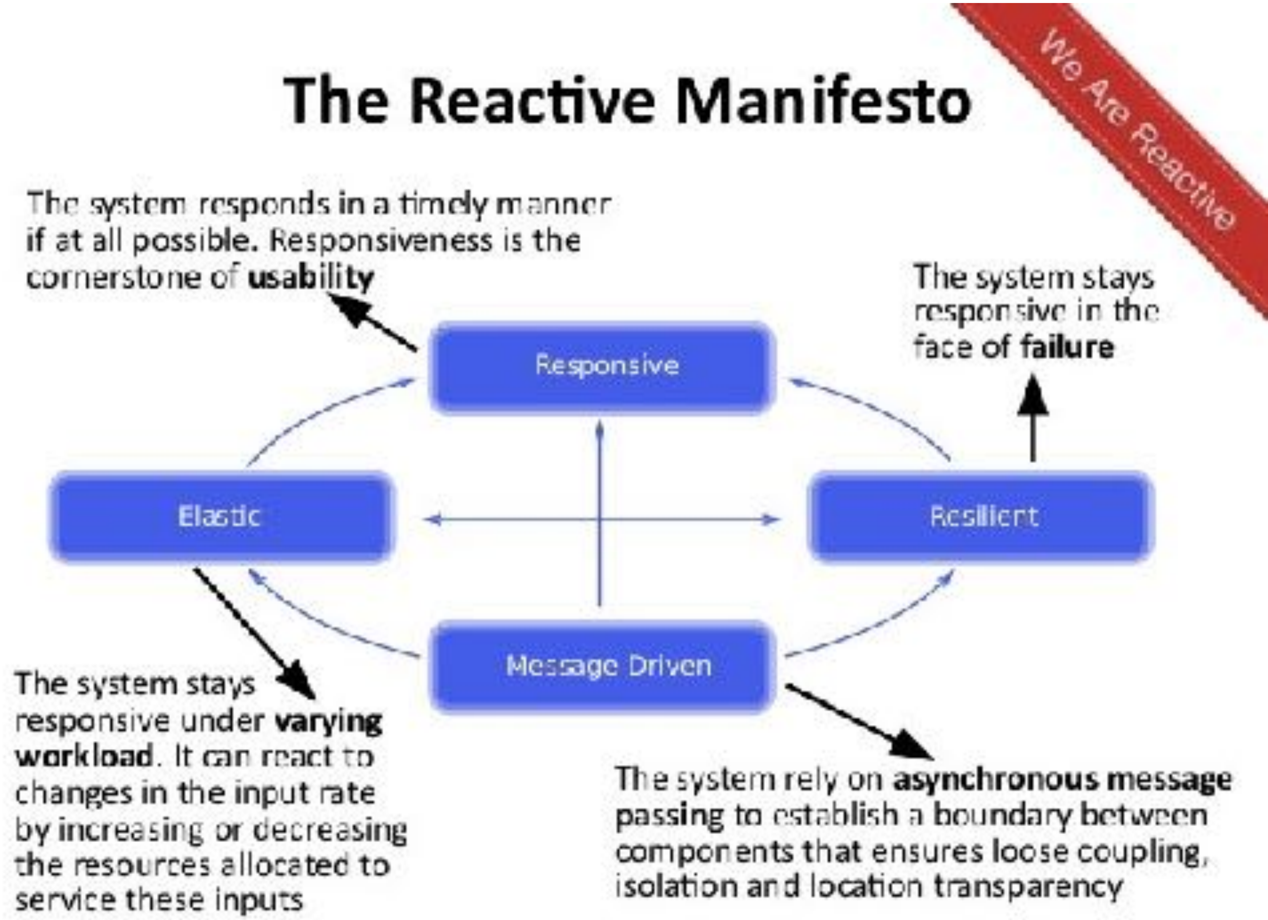
PASSIONATE DEVELOPER, SPEAKER, ACTIVE MEMBER OF KIEV JUG.

PARTICIPATE IN DIFFERENT EDUCATIONAL INITIATIVES, ENGINEERING EVENTS AND JCP/ADOPTJSR PROGRAMS.

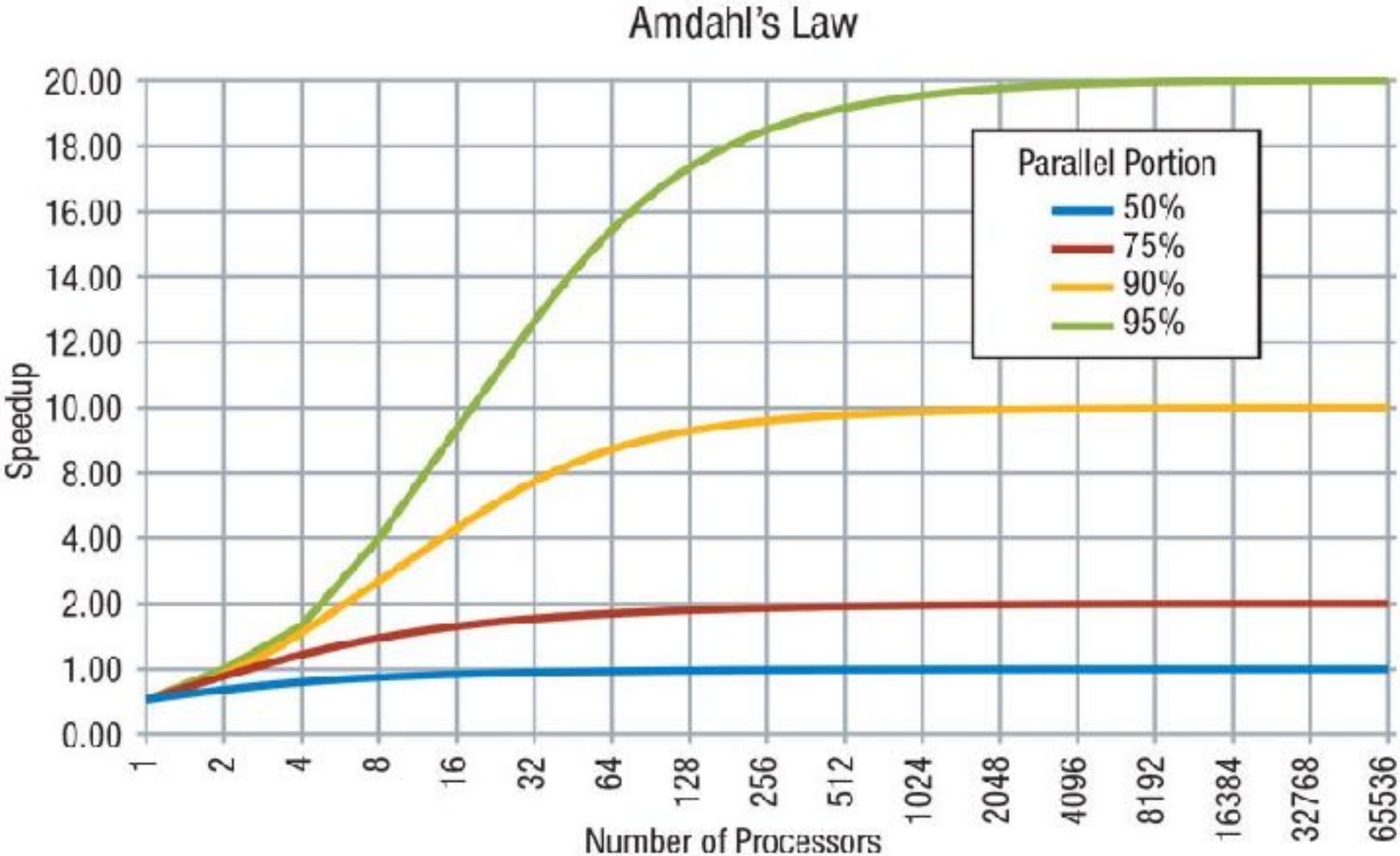
The Reactive Manifesto

Today applications:

- Deployed on everything from mobile devices to cloud-based clusters
- Running thousands of multi-core processors
- Millisecond response times
- 100% uptime




REACTIVE MANIFESTO DOESN'T TELL US HOW TO BUILD SUCH SYSTEMS
NOR PROVIDE ANY INSTRUMENTS FOR THIS
risovach.ru


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
4.00
2.00
1.00
0.00
Parallel Portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

Non-blocking architecture

Programming style change required:

- Can’t write imperative code anymore
- Can’t assume single thread of control
- Must deal with async results (listeners/callbacks)
- Everything becomes stream of events

What is Reactive Programming?

**The basic idea behind reactive programming is that there are certain datatypes that represent a value "over time".
Computations that involve these changing-over-time values will themselves have values that change over time.**

**An easy way of reaching a first intuition about what it's like is to imagine your program is a spreadsheet and all of your variables are cells.
If any of the cells in a spreadsheet change, any cells that refer to that cell change as well.**

Reactive programming

vs

Concurrent programming

- Reactive programming first of all is about reactive data
- Reactive programming helps to better utilize existing resources
- Reactive programming helps to avoid blocking operations using non-blocking IO
- Reactive programming is more about scaling vertically than scaling horizontally
- Reactive programming promotes asynchronous processing but it could be synchronous as well.
- Reactive programming can greatly benefit from concurrency and multithreading however it could be single-threaded where necessary. Instead Reactive Streams provide transparency over threading model used behind scenes.

Reactive Use Cases

- **External Service Calls**
  - Abstract away blocking IO
- **Highly Concurrent Message Consumers**
  - Existing Reactive libraries provide convenient API for building highly concurrent and high performance processing pipelines
- **Spreadsheets**
  - If application should be reactive in nature than it is obvious choice
- **Abstraction Over (A)synchronous Processing**
  - Existing Reactive libraries provide good abstraction from sync/async processing details as well as from threading model behind

Existing implementations

akka

VERT.X

Reactive Streams spec

Reactive Streams spec

Reactive Streams is an initiative to provide a standard for **asynchronous stream processing** with **non-blocking back pressure**.

The Problem

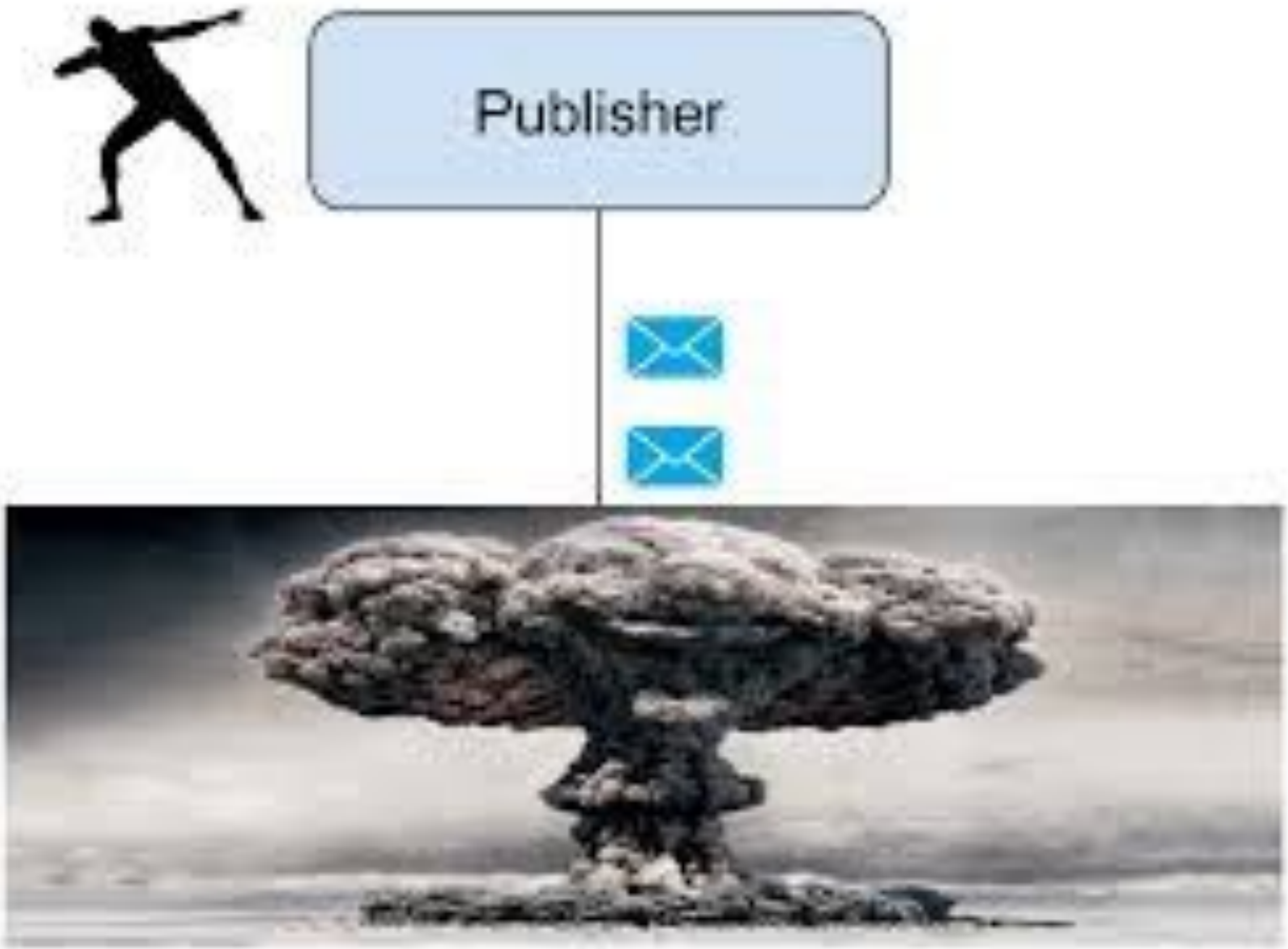
Handling **streams of data**—especially **“live” data** whose **volume is not predetermined**—**requires special care** in an asynchronous system.

The **most prominent issue** is that resource consumption needs to be controlled such that a **fast data source does not overwhelm the stream destination**.

Scope

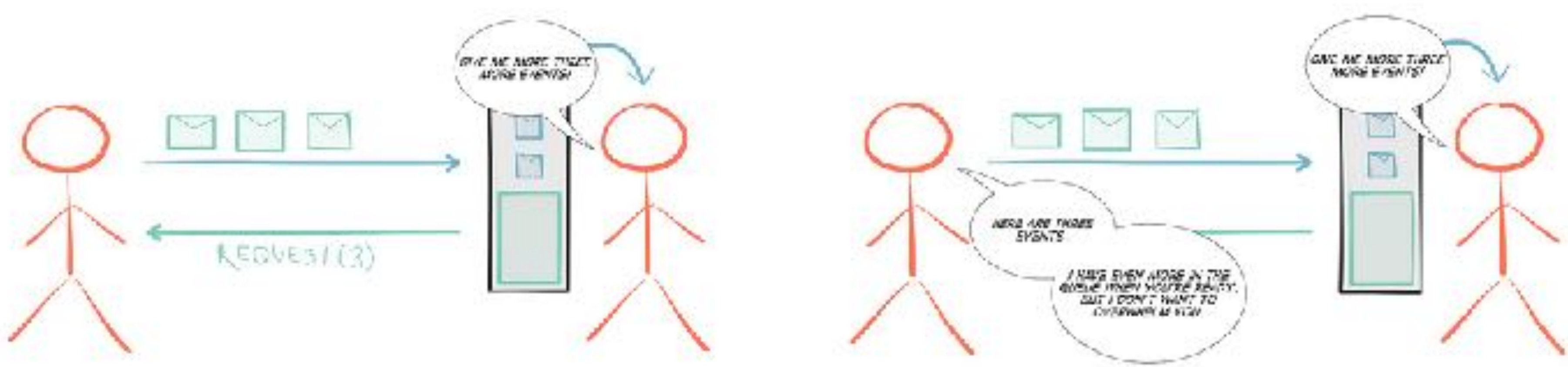
The scope of Reactive Streams is to find a **minimal set of interfaces**, **methods and protocols** that will describe the necessary operations and entities **to achieve the goal—asynchronous streams of data with non-blocking back pressure**.

What is back-pressure?



Back-pressure

- When fast publisher pushing messages to slow consumer queues can overflow
- Back-pressure allows subscriber to specify/control demand:
  - subscriber requests # of items
  - publisher produce up to requested items
- If data source is hot and can’t be controlled publisher should have particular policy in place: it might buffer or drop messages on back-pressure



Reactive Streams API

```java
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> var1);
}
```

```java
public interface Subscriber<T> {
    void onSubscribe(Subscription var1);

    void onNext(T var1);

    void onError(Throwable var1);

    void onComplete();
}
```

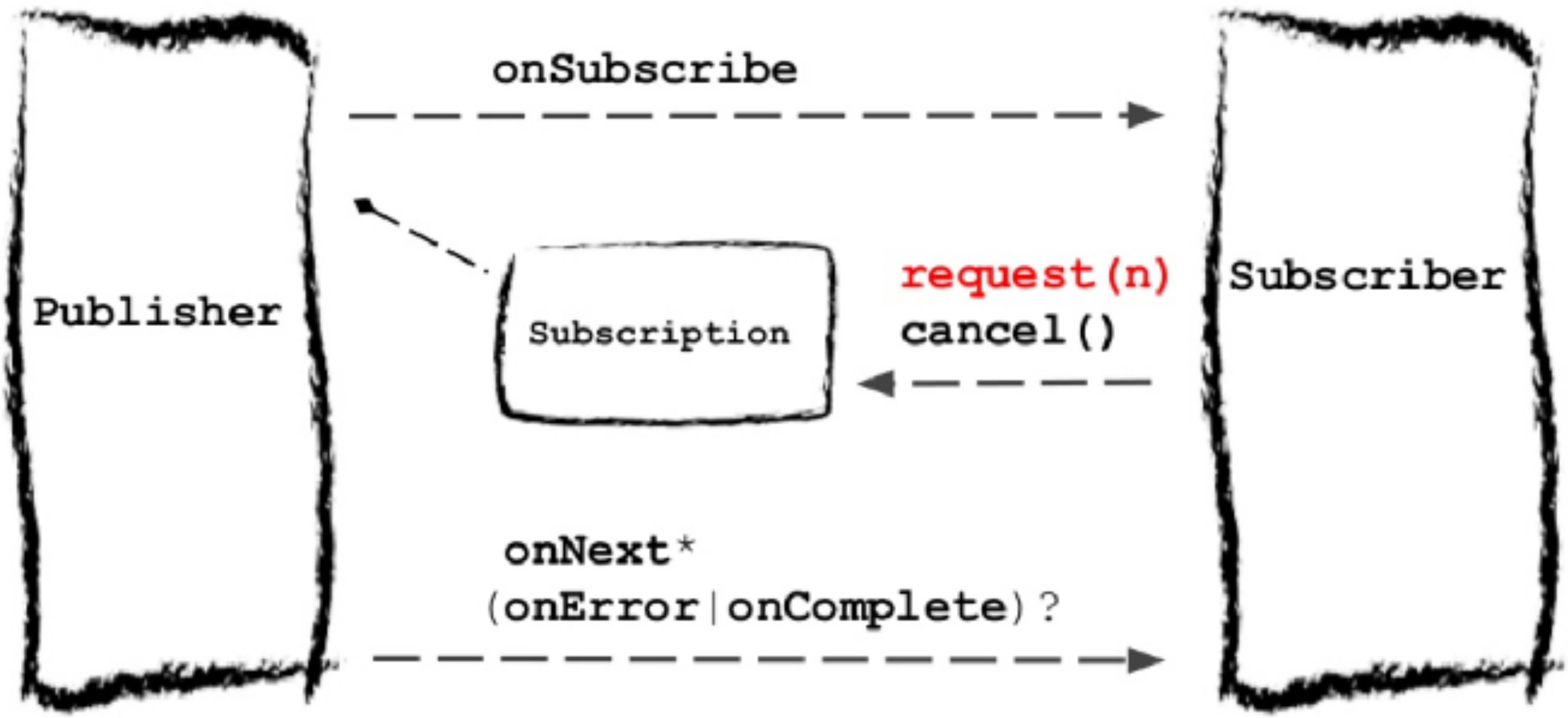
```java
public interface Subscription {
    void request(long var1);

    void cancel();
}
```

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

Publish-Subscribe Flow

Reactive Streams API



Difference to CompletableFuture

`CompletableFuture` is *push* only model.
If you have a reference to the **`Future`**,
it means the task processing an asynchronous result is already executing!

ReactiveStreams enable ***deferred pull-push*** interaction:

- ***Deferred*** because nothing happens before the call to **`subscribe()`**
- *Pull* because at the subscription and request steps, the **`Subscriber`** will send a signal upstream to the source and essentially ***pull*** the next chunk of data
- ***Push*** from producer to consumer from there on, within the boundary of the number of requested elements

Reactive libraries generations

- **0th generation**: Publish-subscribe based on Observer pattern (Swing/AWT/Android)
- **1st generation**: Rx.NET around 2010, Reactive4Java in 2011 and early versions of RxJava in 2013 (backpressure is not supported, synchronous cancellation issue)
- **2nd generation**: RxJava redesigned (backpressure support and operators composition)
- **3rd generation:** Reactive-Streams specification compatible libs (RxJava 2.x, Project Reactor and Akka-Streams)
- **4th generation:** Fluent API with diff optimisations like operator-fusion (RxJava 2.x redesigned, Project Reactor 2.5+ and eventually Akka-Streams)
- **5+ generation:** Reactive IO, transparent remote reactive queries and more…

Operator-fusion

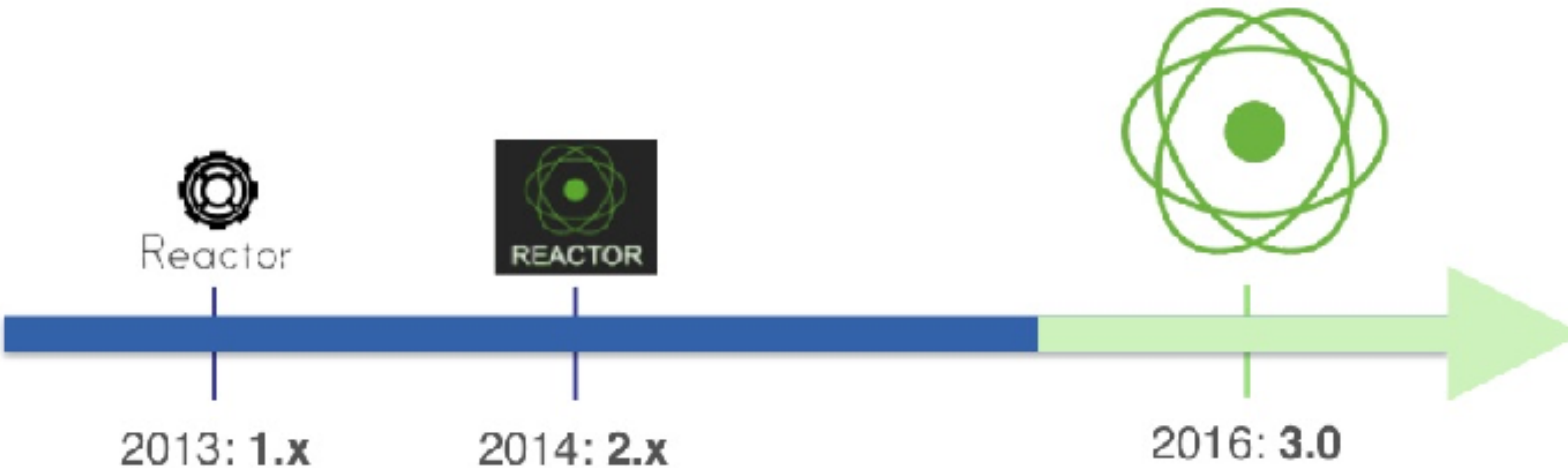
Operator-fusion, one of the cutting-edge research topics in the reactive programming world, is the aim to have two of more subsequent operators combined in a way that reduces overhead (time, memory) of the dataflow.

2 key categories :

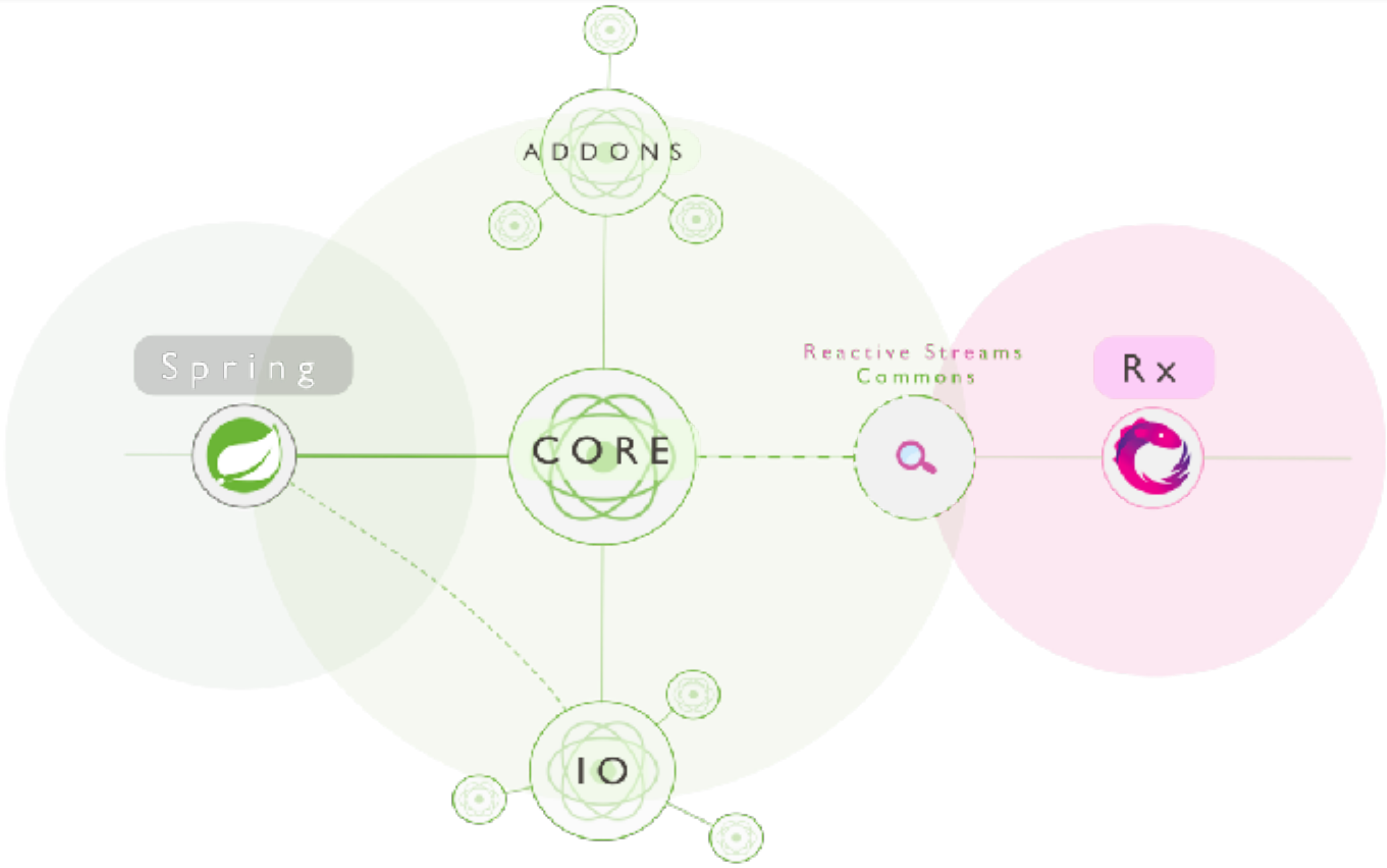
- **“Macro Fusion”** : merge operators in one (assembly time)
- **“Micro Fusion”** : avoid queue creation and short circuit where possible request lifecycle.

Spring Reactor

Spring Reactor evolution



Reactor in Spring ecosystem



It’s a simple Flux, nothing to worry about...

```java
Flux<Integer> result = Flux.just(1, 2, 3);

Mono<String> mono = Mono.just("Result");
```

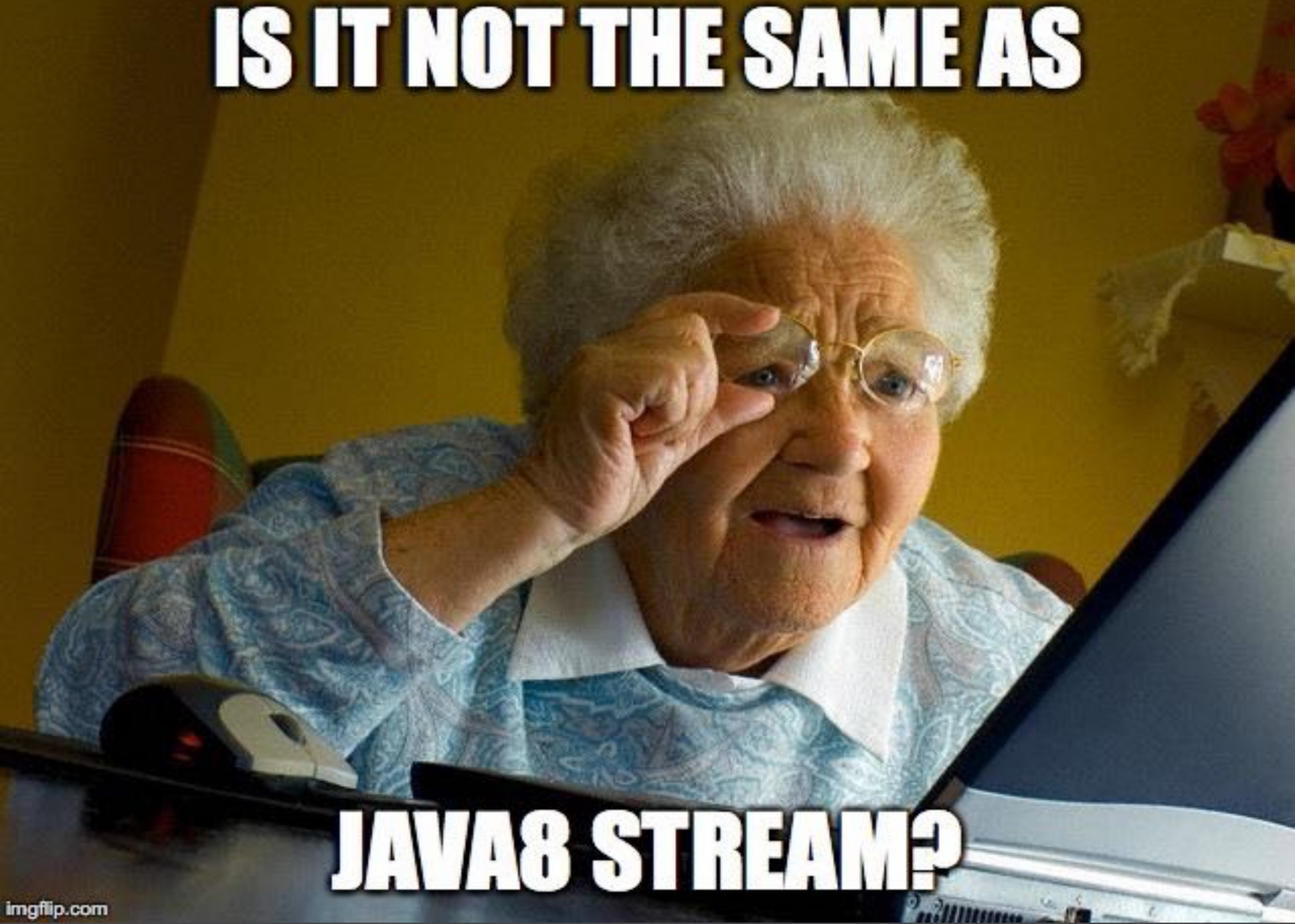
There are basically three things you can do with Flux/Mono:

- **operate on it** (transform it, or combine it with other sequences)
- **subscribe to it** (it’s a publisher)
- **configure it** (modify the behaviour of subscribers)

Simple example

```
Flux.just(1, 2, 3)
            .filter(num -> num % 2 != 0)
            .map(String::valueOf)
            .log()
            .subscribe(System.out::println);
```

```
INFO: | onSubscribe([Synchronous Fuseable] FluxMapFuseable.MapFuseableSubscriber)
May 16, 2017 5:56:50 PM reactor.util.Loggers$JdkLogger info
INFO: | request(unbounded)
May 16, 2017 5:56:50 PM reactor.util.Loggers$JdkLogger info
INFO: | onNext(1)
May 16, 2017 5:56:50 PM reactor.util.Loggers$JdkLogger info
INFO: | onNext(3)
May 16, 2017 5:56:50 PM reactor.util.Loggers$JdkLogger info
INFO: | onComplete()
1
3
```


IS IT NOT THE SAME AS
JAVA8 STREAM?
imgflip.com

Reactive Streams
vs
Java8 Streams

- Programming model is very similar
- Both heavily use operators chaining and lambdas

- However purpose is different!!!
- Reactive streams operate on reactive data and represent data over time
- Java8 streams operate on collections and have all or nothing semantics

VS

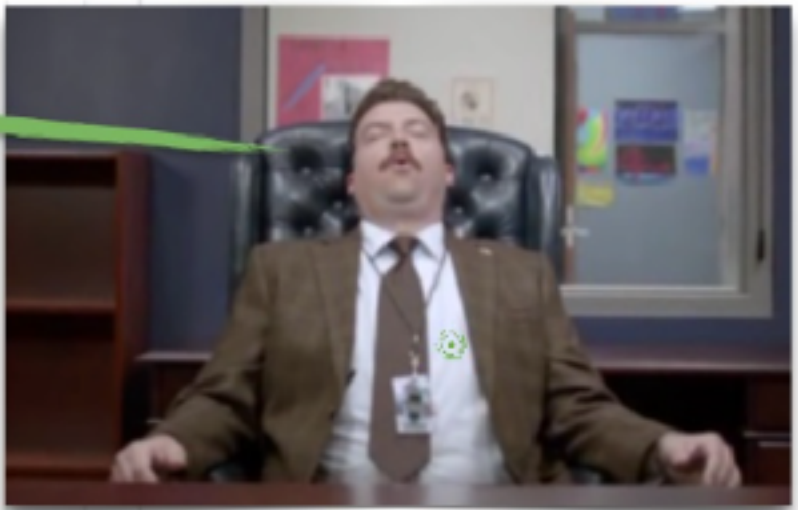
Reactive streams lifecycle

```
Flux.just(1, 2, 3)
        .filter(num -> num % 2 != 0)
        .map(String::valueOf)
        .log()
        .subscribe(System.out::println);
```

- **Assembly-time:** This is the time when you write up `just().subscribeOn().map()`
- **Subscription-time:** This is the time when a Subscriber subscribes to a sequence at its very end and triggers a "storm" of subscriptions inside the various operators.
- **Runtime:** This is the time when items are generated followed by zero or one terminal event of error/completion

ParallelFlux

Efficient and micro bench ready



https://github.com/akarnokd/akarnokd-misc/tree/master/src/jmh/java/hu/akarnokd/comparison

Join JavaDay Kyiv 2017

Discount code

25Jug

Links

Exercises to practice: https://github.com/reactor/lite-rx-api-hands-on
Examples shown: https://github.com/olegts/ReactiveStreams/tree/master/src/test/java/org/reactivestreams/reactor/javaday

Reactive Streams: https://github.com/reactive-streams/reactive-streams-jvm
Project Reactor docs: https://projectreactor.io/docs