Introduction to Reactive Streams and Project Reactor

Zoltan Altfatter

Why going Reactive?

- Hype or reality?
- Do I need to change?
- Imperative programming model involving blocking operations it is difficult to scale in an efficient manner
- With Reactive programming the promise is that we can do more with less, specifically you can process higher workloads with fewer threads
- For the right problem, the effect can be dramatic, for the wrong problem the effects might go into reverse

Early reactive solutions

- **Future** – is asynchronous but blocks current thread with the get() method
- **ListenableFuture** – Spring 4 type a Future implementation that adds the capability to accept completion callbacks
- **CompletableFuture –** defines the contract for an asynchronous computation step that can be combined with other steps, no support for multiple items
- **Stream API –** not designed for operations with latency, such as I/O, it is pull based only, can only be used once

Reactive programming

- Reactive programming engage in the concept of **non-blocking asynchronous** operations
- **non-blocking** -> don’t want to hold on to a thread when I am not actually using it (concept of event-loop)
- **asynchronous** -> the answer comes later, whether by polling or by an event pushed back to us

Side effect -> applications can accomplish more with existing resources

Reactive programming

- Reactive programming is the next frontier in Java for high-efficiency applications
- Key differentiator form “async” is Reactive (pull-push) **back pressure**

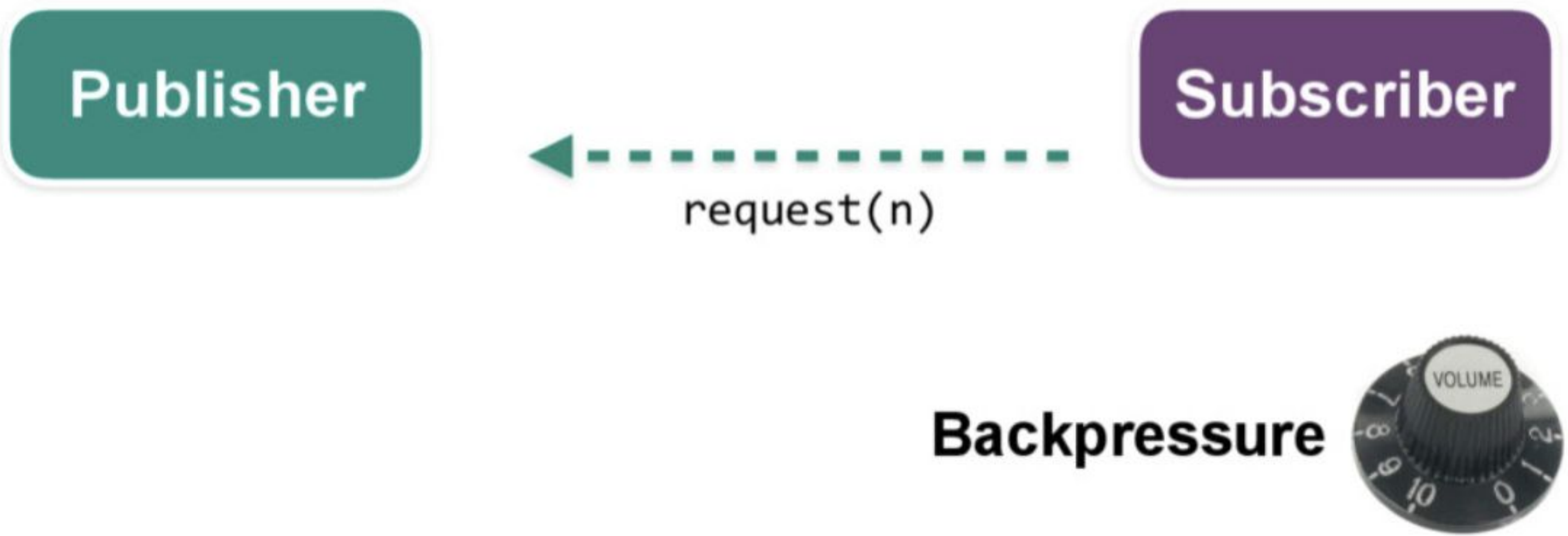
Reactive Streams

- Manifest: http://www.reactive-streams.org
- Standard created as of industry collaboration (included in Java 9 as Flow API)
- Provides interoperability between **Reactive Streams Libraries**
- Introduces the concept of **back pressure** (volume control)
- The consumer controls how much data is sent using a pull-based mechanism instead of a traditional push-based mechanism

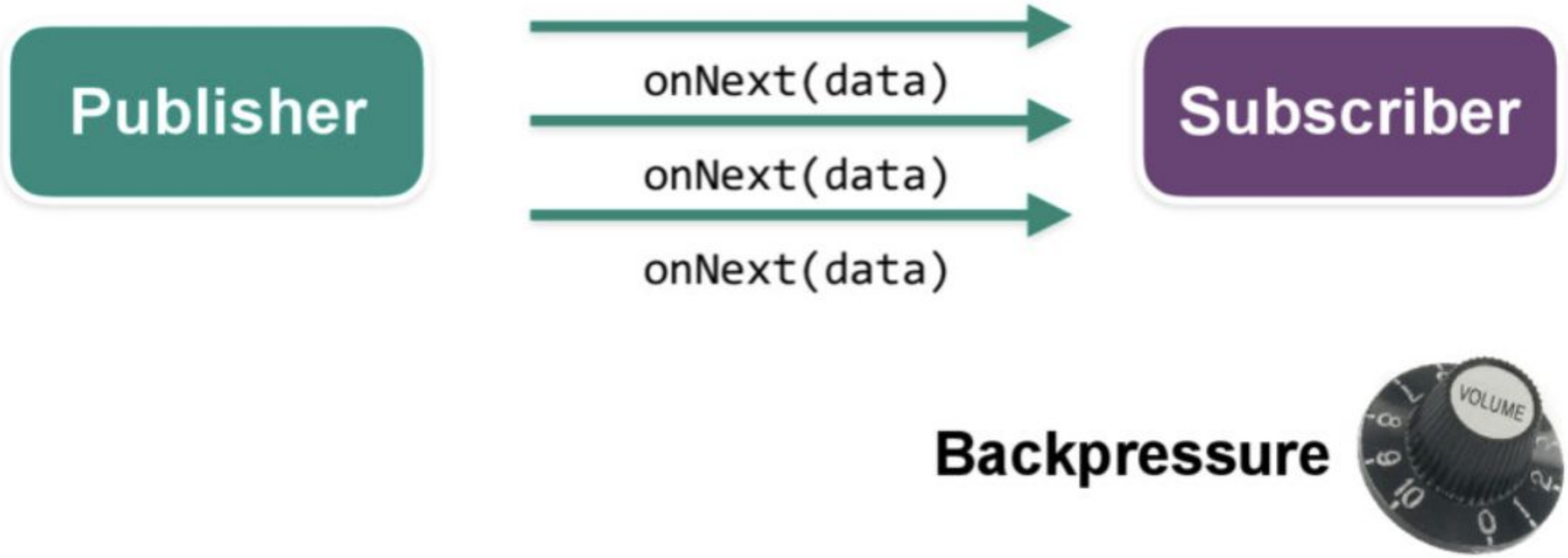
Reactive Streams

Reactive Streams

Reactive Streams



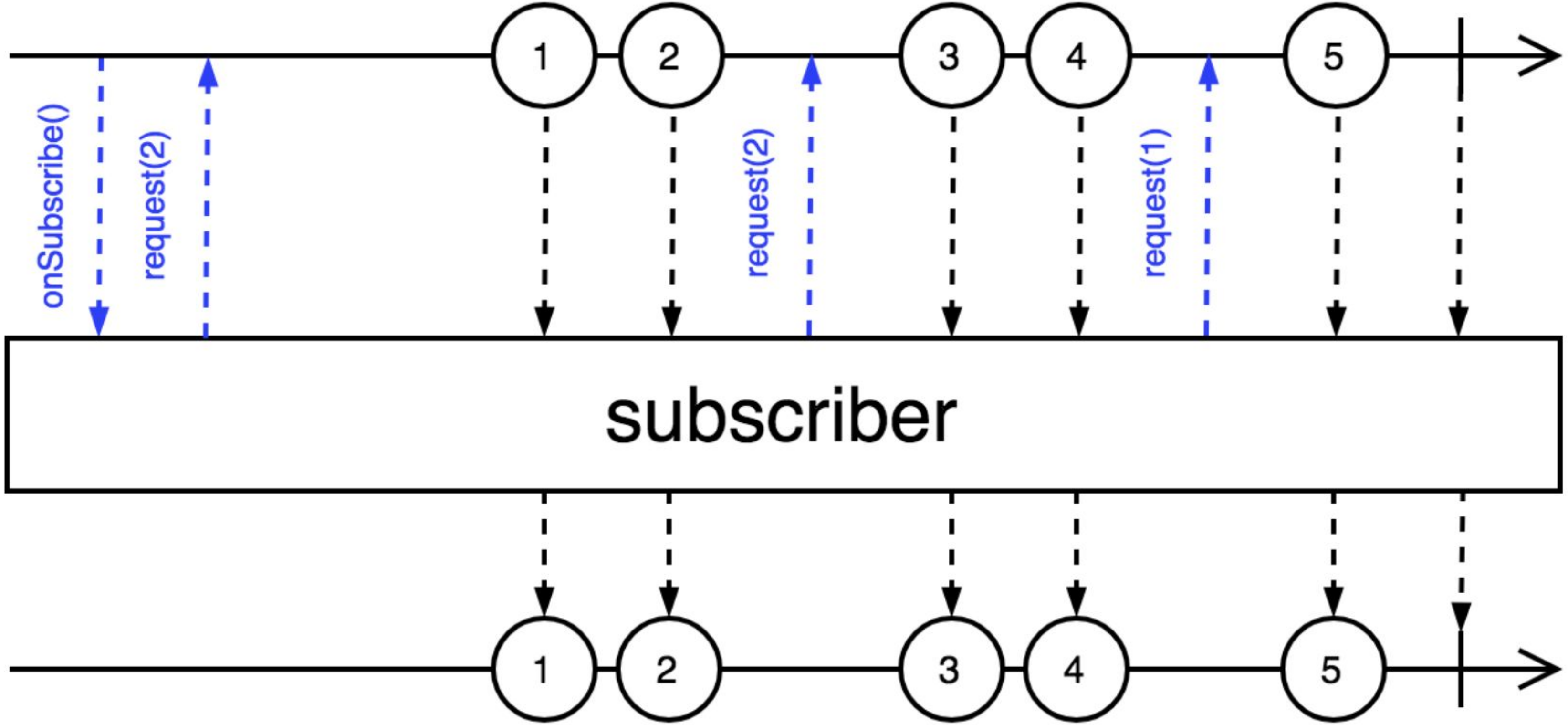
Reactive Streams



Reactive Streams

Backpressure

Back pressure



Reactive Streams API

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

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

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

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

Reactive Streams libraries

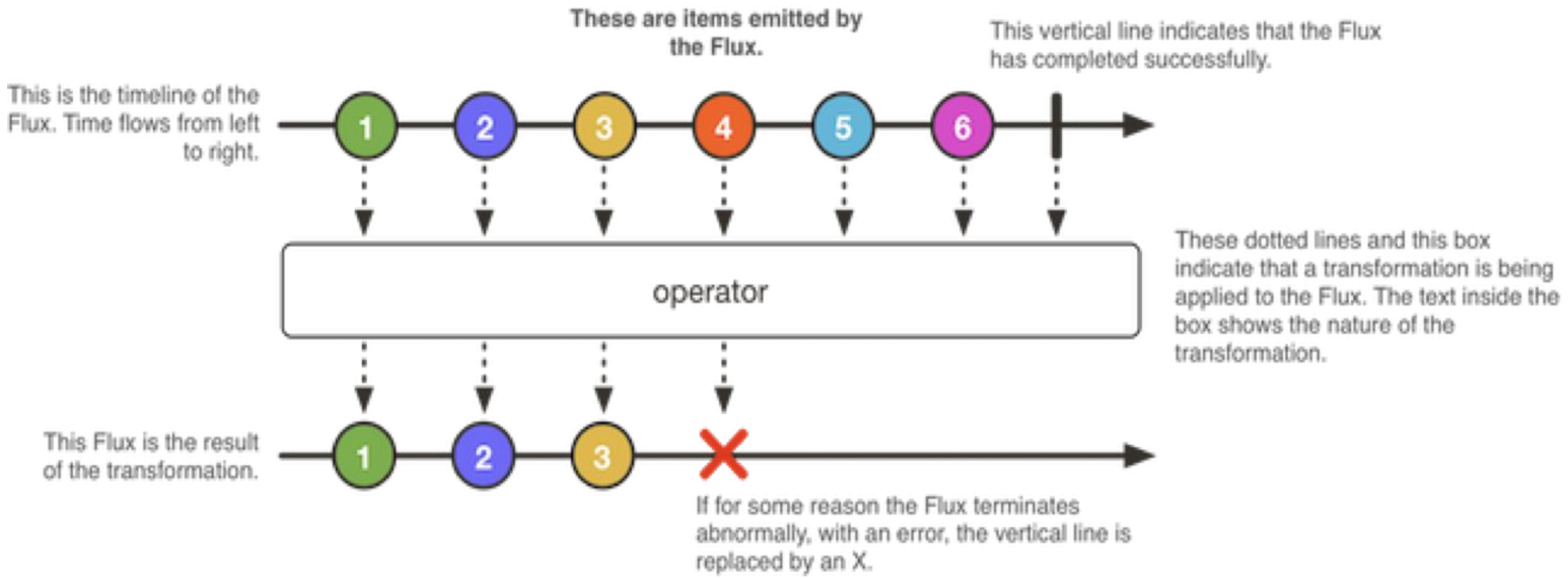
- Reactive Streams specification is too low level to work with
- Reactive Libraries implement Reactive Streams
- Reactive Libraries provide additional types

RxJava Reactor Akka Streams

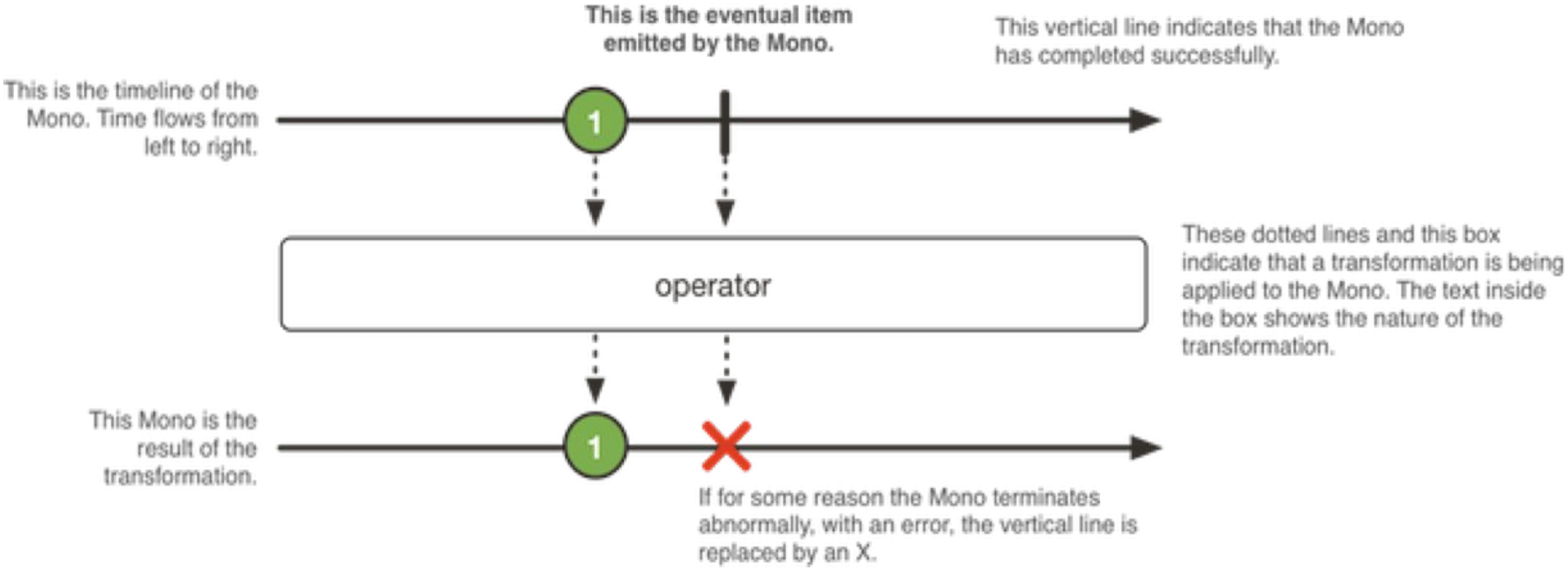
Project Reactor

- https://projectreactor.io/
- Implements Reactive Streams specification
- Provides **Mono** and **Flux** API types
- **Mono** - 0..1 elements
- **Flux** - 0..N elements
- Provides many operators on these types which support **non-blocking back pressure.**
- Spring WebFlux requires Reactor as core dependency, but it is interoperable with other reactive libraries via Reactive Streams

Flux<T> is a Publisher<T> for 0..n elements

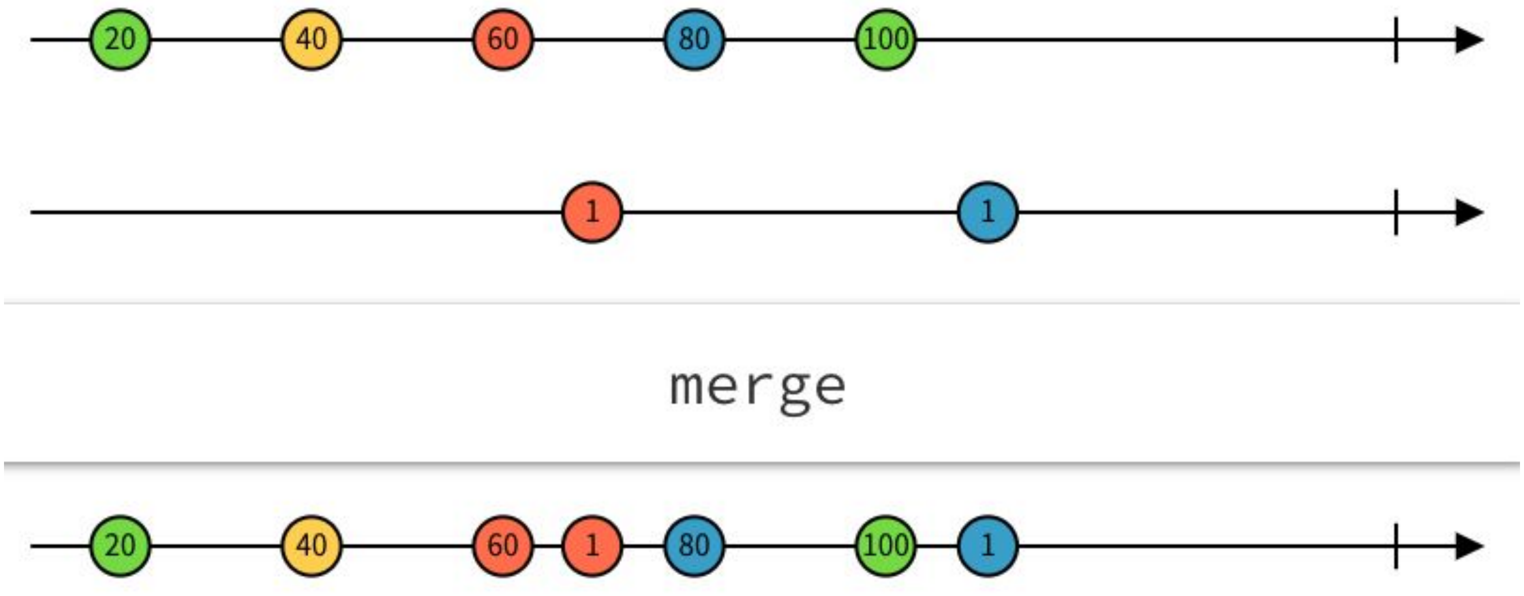


Mono<T> is a Publisher<T> for 0..1 elements

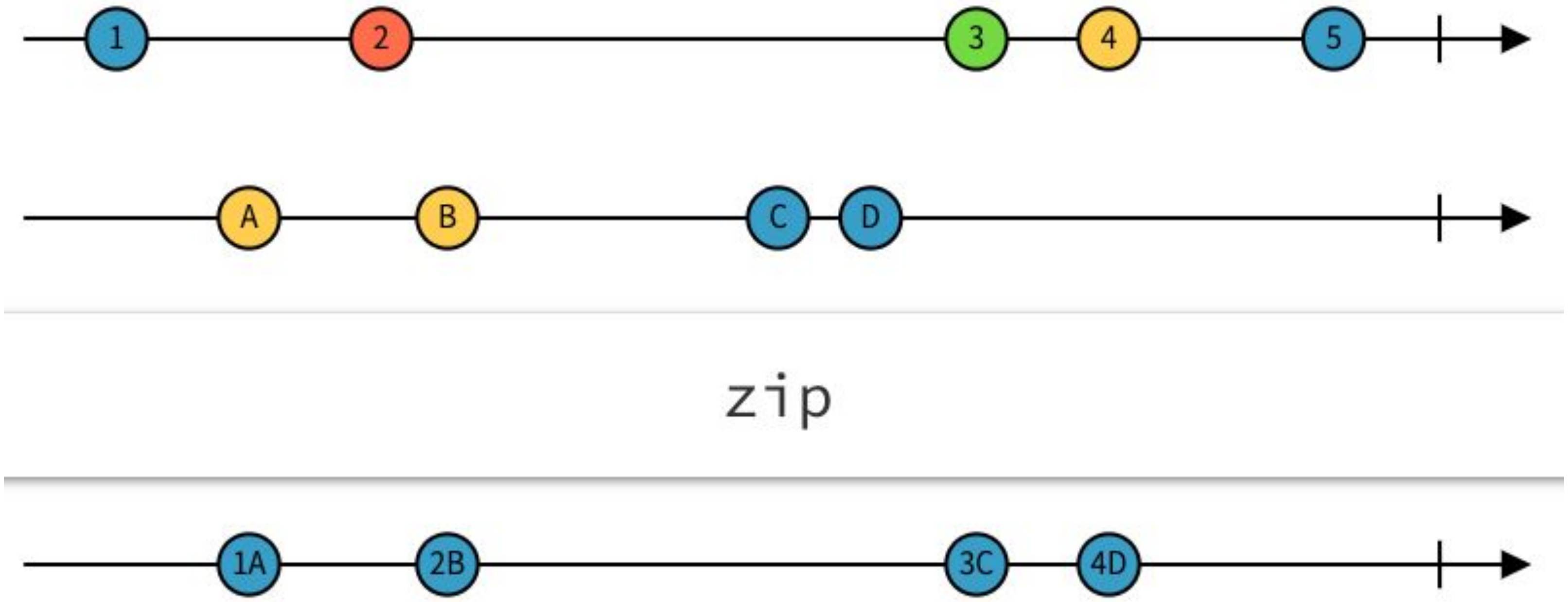


Marble Diagrams

Interactive marble diagrams http://rxmarbles.com/



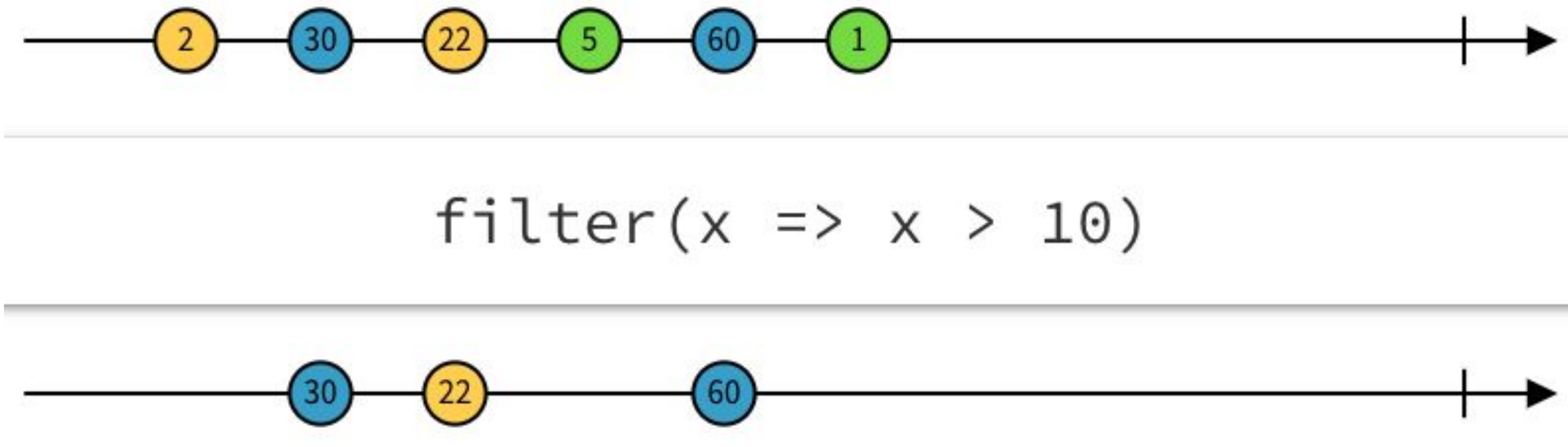
Combine streams



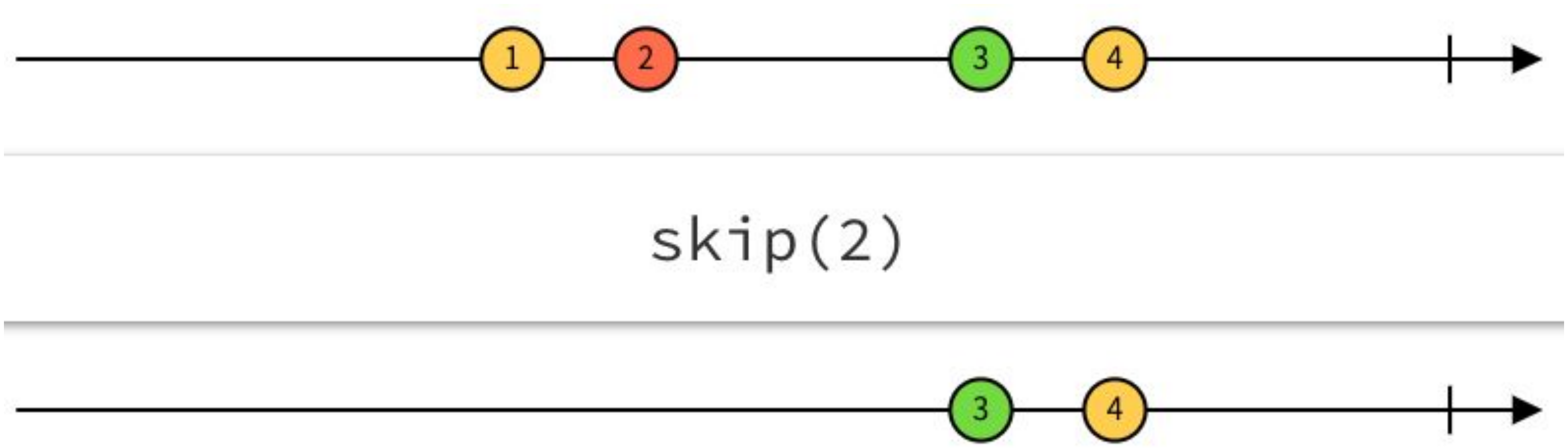
Create operators

```
Flux.just(value)

Flux.fromIterable(list)

Flux.range(i, n)

Flux.interval(Duration.ofSeconds(n))

...
```

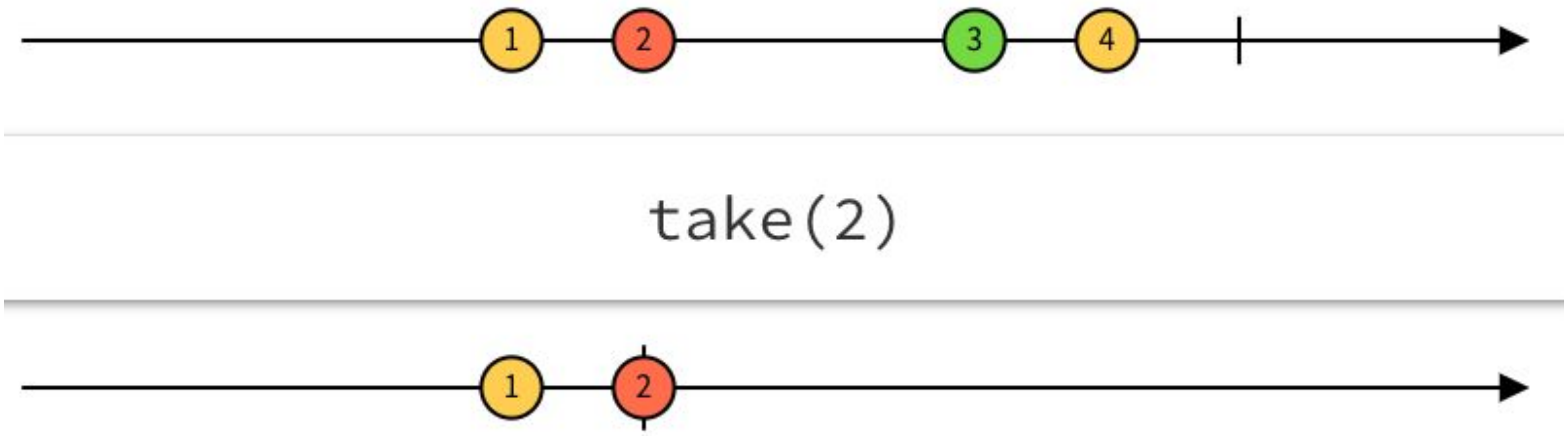
Transform operators



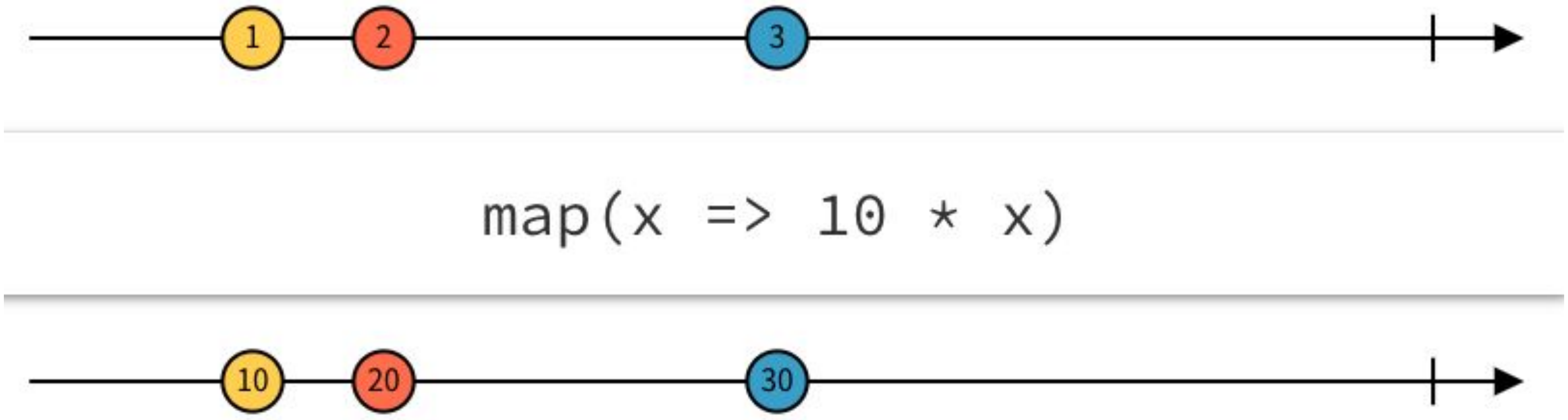
Transform operators



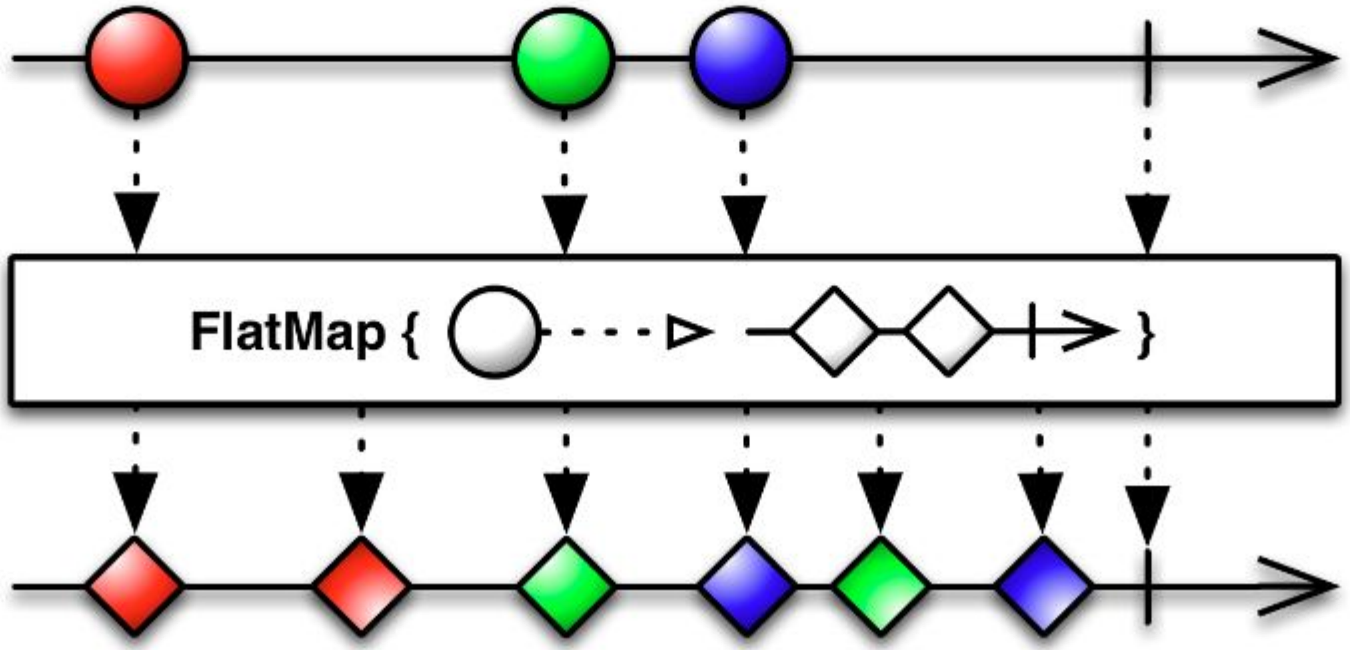
Transform operators



Transform operators



Transform operators



Difference between map and flatMap

- **map** operator
  - transforms **1** source element into **1** output element
  - does nothing particular other than transformation from **T** to **V**
  - transformation is **synchronous**
  - returns a **Flux<V>**
- **flatMap** operator
  - transforms **1** source element into a **Flux of N** elements
  - subscribes to each generated **Flux<V>** then **flattens** their values
  - transformation can be **async**
  - returns a **Flux<V>**

Java Stream vs Reactor APIs

- Java Stream API
  - functional-style API to process a collection **in-memory** data once.
- Reactor API
  - functional-style API to process any sort of data (including asynchronously generated data, possibly multiple times

Java Stream vs Reactor APIs

• Java Stream
  ○ **pull** based
  ○ a way to iterate collections declaratively
  ○ generally **synchronous** data
  ○ streams can be used only **once**
  ○ **no flow-control**
  ○ **no composition of streams**
  ○ **finite** amount of data
	• **Flux / Mono**
	  ○ **push** based
	  ○ real-time data, latency, flow control
	  ○ **asynchronous** friendly
	  ○ **back pressure**
	  ○ **advanced composition / transformation**
	  ○ data sizes from **zero** to **infinity**

Nothing happens until you subscribe

- Until you subscribe what you do is that you **declare a processing pipeline**, what you intend to do as soon as the data is becoming available

```
Flux.range(0, 10)
    .map(i -> "got " + i)
    .subscribe();


Flux.range(0, 10)
        .map(i -> "got " + i)
        .subscribe(
                value -> handleHappyPath(value),
                error -> handleError(error),  // optional but recommended
                () -> handleCompletion());    // optional


Flux.range(0, 10)
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) {
                System.out.println(integer);
            }
        });
```

Hot vs Cold Publisher

- a **Cold** Publisher (like on the previous slide) re-generates the data again if you subscribe to it again
- A **Hot** Publisher does not start from scratch for each Subscriber.

```
Flux<Integer> flux = Flux.range(0, 4);

ConnectableFlux<Integer> cflux = flux.delayElements(Duration.ofMillis(400)).publish();
cflux.subscribe(i -> System.out.println("first received: " + i + " on thread " +Thread.currentThread().getName()));

cflux.connect();

Thread.sleep(1000);
cflux.subscribe(i -> System.out.println("second received: " + i + " on thread " + Thread.currentThread().getName()));
```

```
first received: 0 on thread parallel-1
first received: 1 on thread parallel-2
first received: 2 on thread parallel-3
second received: 2 on thread parallel-3
first received: 3 on thread parallel-4
second received: 3 on thread parallel-4
```

publishOn operator

```
Flux.range(0, 2)
       .map(i -> {
           System.out.printf("processing element %d on thread %s%n", i, Thread.currentThread().getName());
           return i*2;
       })
       .subscribe(i -> System.out.printf("got element %d on thread %s%n", i, Thread.currentThread().getName()));

processing element 0 on thread main
got element 0 on thread main
processing element 1 on thread main
got element 2 on thread main

Flux.range(0, 2)
       .map(i -> {
           System.out.printf("processing element %d on thread %s%n", i, Thread.currentThread().getName());
           return i*2;
       })
       .publishOn(Schedulers.elastic())
       .subscribe(i -> System.out.printf("got element %d on thread %s%n", i, Thread.currentThread().getName()));

processing element 0 on thread main
processing element 1 on thread main
got element 0 on thread elastic-2
got element 2 on thread elastic-2
```

subscribeOn operator

```
Flux.range(0, 2)
        .map(i -> {
            System.out.printf("processing element %d on thread %s%n", i, Thread.currentThread().getName());
            return i*2;
        })
        .subscribeOn(Schedulers.elastic())
        .subscribe(i -> System.out.printf("got element %d on thread %s%n", i, Thread.currentThread().getName()));
```

```
processing element 0 on thread elastic-2
got element 0 on thread elastic-2
processing element 1 on thread elastic-2
got element 2 on thread elastic-2
```

subscribeOn - changes the thread where the source starts to generate data

https://zoltanaltfatter.com/2018/08/26/subscribeOn-publishOn-in-Reactor/

VirtualTime

- Scheduler abstraction
  - `Schedulers.elastic`
  - `Schedulers.parallel`
- Allows time travel
- Switches the Scheduler under the hood

```
StepVerifier.withVirtualTime(() -> Flux.range(0, 4).delayElements(Duration.ofHours(2)))
        .expectSubscription()
        .thenAwait(Duration.ofDays(1))
        .expectNextCount(4)
        .verifyComplete();
```

Reactor Core Workshop

https://github.com/altfatterz/spring-reactive-workshop

Modules:

- **reactor-core-workshop**
- **reactor-core-workshop-solution**