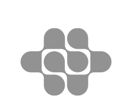
Reactive Programming



About Me

Sasa Velickovic
sasa.velickovic@endava.com

Design Authority in Transport & Logistics
Solution Architect
Distributed Systems and Integrations (Patterns)
Software Engineering Background

Reactive Programming

- Reactive Programming is a programming paradigm concerned with data streams and the propagation of change https://en.wikipedia.org/wiki/Reactive_programming
- Type of dataflow programming, traces back to 1970 and through the years saw its primary usage in implementing user interfaces.
- With Reactive Programming it is possible to express static and dynamic data streams.
- Functional Reactive Programming

IMPERATIVE

```
b = 1
c = 2
a = b + c
b = 10

print a : 3
```

REACTIVE

```
b = 1
c = 2
a -> b + c
b = 10

print a : 12
```

 © COPYRIGHT 2021 ENDAVA

Reactive Streams Specification

Reactive Streams is a Functional Reactive Programming model for processing asynchronous data streams.

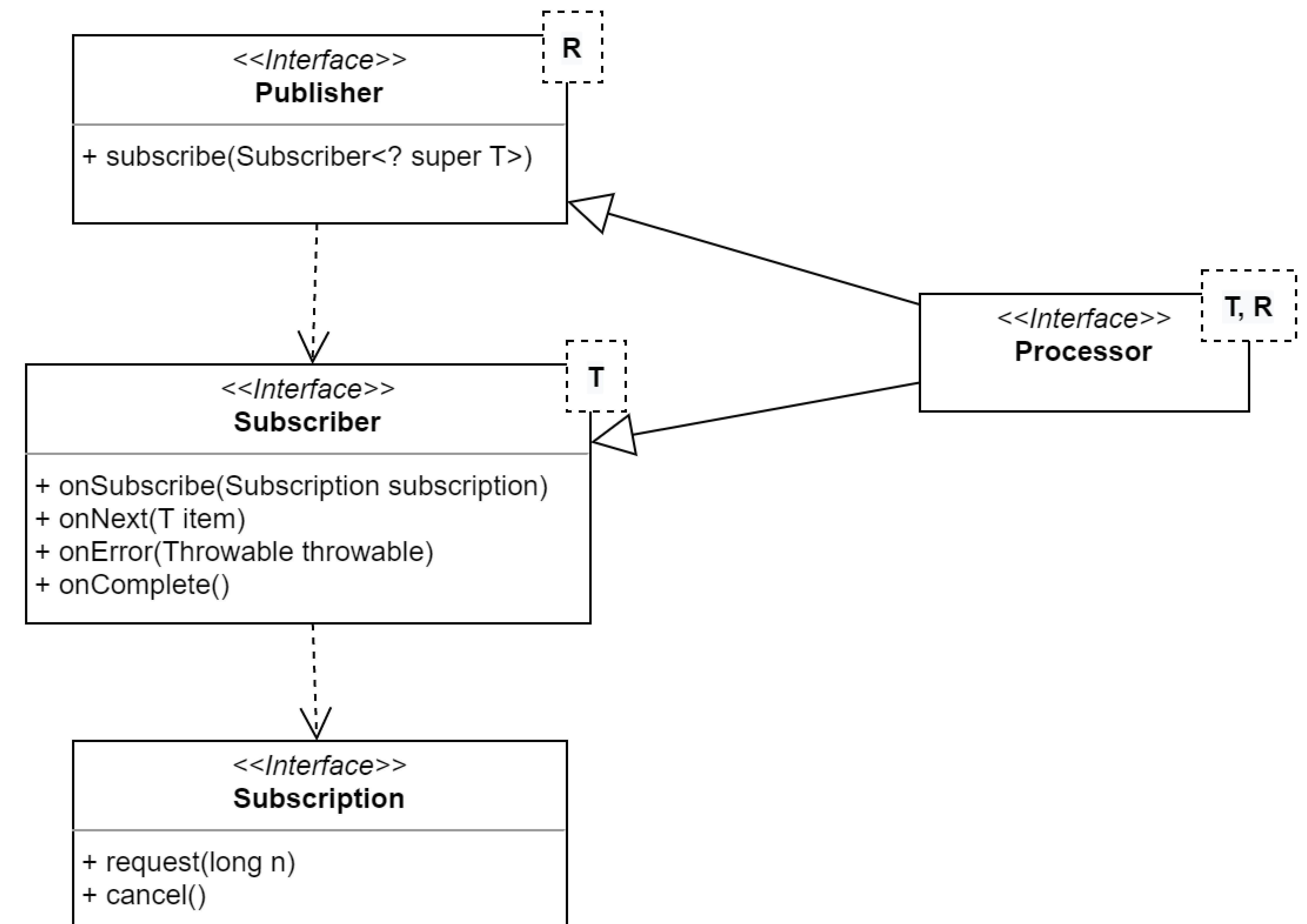
Back pressure - subscriber can use subscription object to specify much it can consume

Concurrent agnostic – threading model is not imposed

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments as well as network protocols.
https://www.reactive-streams.org/
https://github.com/reactive-streams

Included in Java 9 as part of core concurrent libraries
https://openjdk.java.net/jeps/266

Implementations: Akka Streams, Ratpack, Project Reactor, ReactiveX, Vert.x



© COPYRIGHT 2021 ENDAVA

The Reactive Manifesto

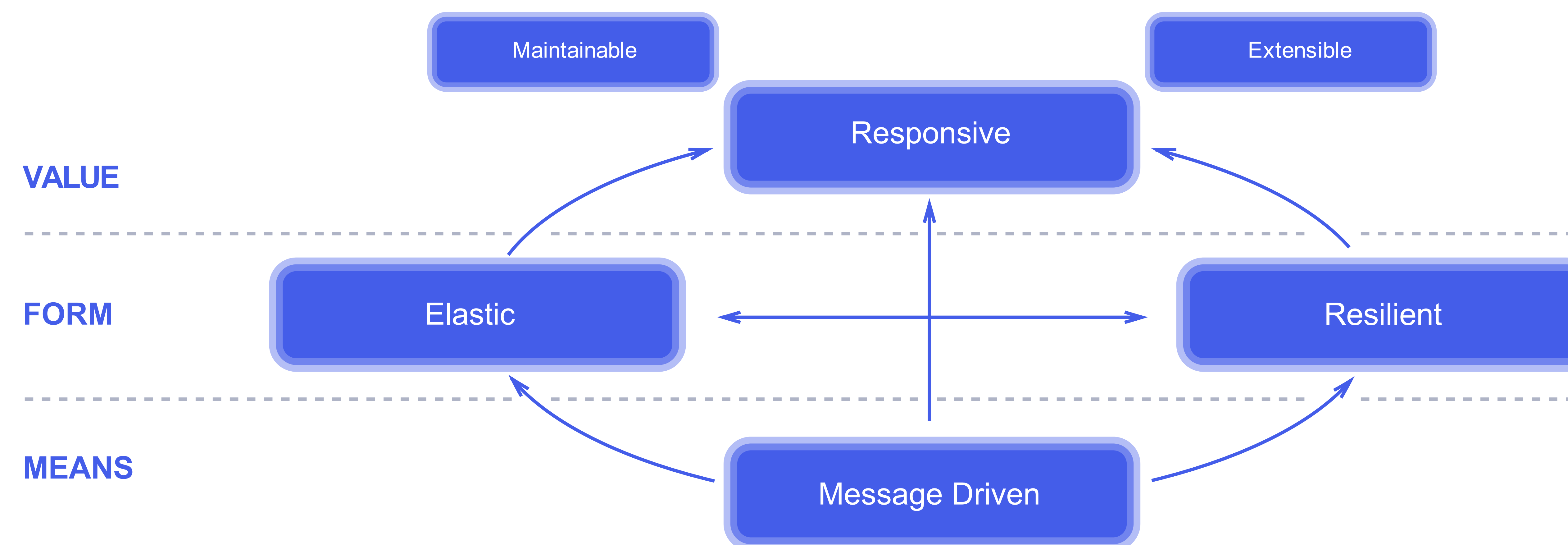
Application requirements have changed dramatically in recent years

- High availability requirements 99.99%
- Milliseconds response times
- Big Data

Reactive Systems are :

- Responsive
- Resilient
- Elastic
- Message Driven

https://www.reactivemanifesto.org/



Flavours of Reactive Programming

REACTIVE (JAVA)

```
public Flux<User> getUsers() {

    return userService.getUsers();
}

public Mono<User> getUserById(String userId) {
    return userService.getUserById(userId);
}
```

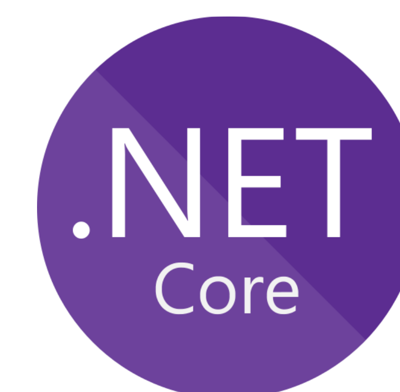
PROMISE (C#)

```
public async Task<IEnumerable<User>> GetUsers()
{
    return await _userService.GetUsers()
}
```

COROUTINES (KOTLIN)

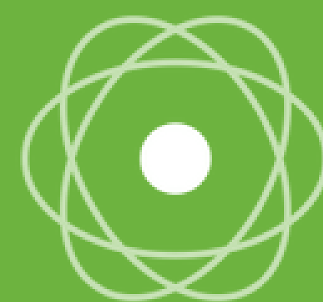
```
fun getUsers(): Flow<User> = userService.getUsers()

suspend fun getUserById(id: String): User? = userService.getUserById(id)
```



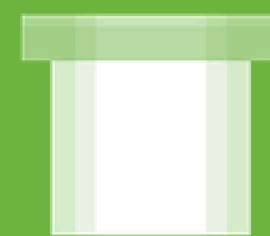
 © COPYRIGHT 2021 ENDAVA

Project Reactor



REACTIVE CORE

Reactor is **fully non-blocking** and provides efficient demand management. It directly interacts with Java's functional API, `CompletableFuture`, `Stream`, and `Duration`.



TYPED [0|1|N] SEQUENCES

Reactor offers **two reactive and composable APIs,** Flux [N] and Mono [0|1], which extensively implement Reactive Extensions.

NON-BLOCKING IO

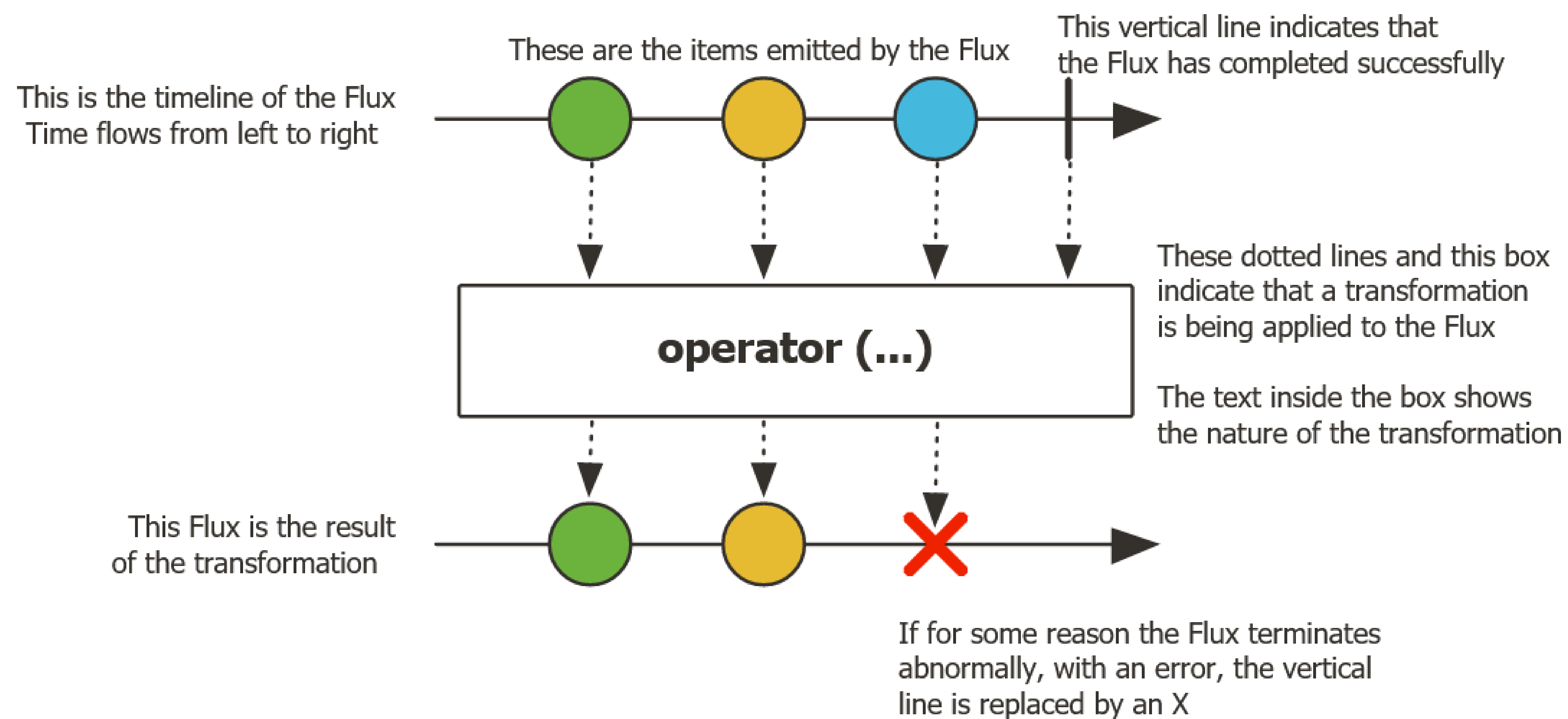
Well-suited for a **microservices** architecture, Reactor offers **backpressure-ready network engines** for HTTP (including Websockets), TCP, and UDP.

https://projectreactor.io/

 © COPYRIGHT 2021 ENDAVA

Project Reactor Flux

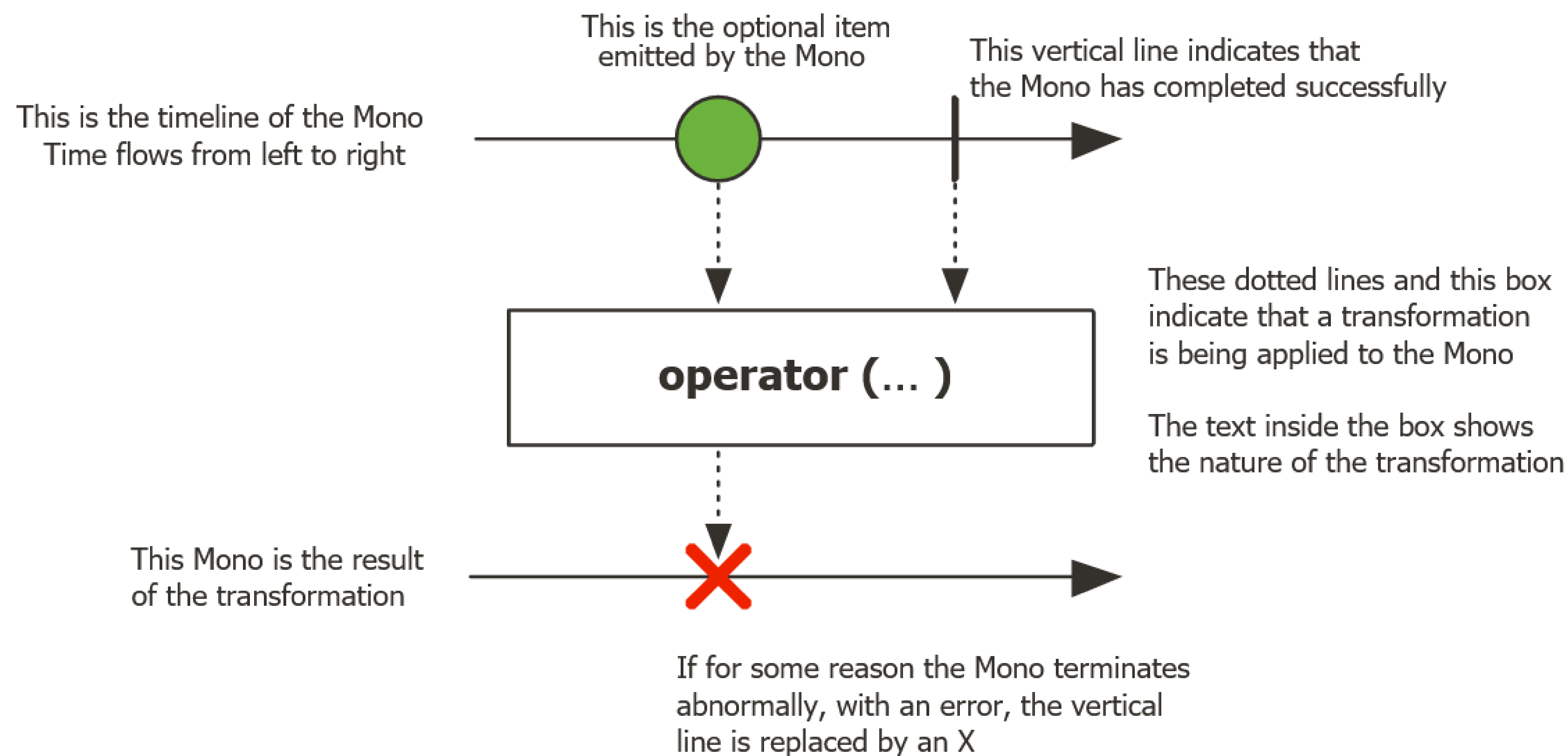
Flux - A Reactive Streams Publisher with rx operators that emits 0 to N elements, and then completes (successfully or with an error).



© COPYRIGHT 2021 ENDAVA

Project Reactor Mono

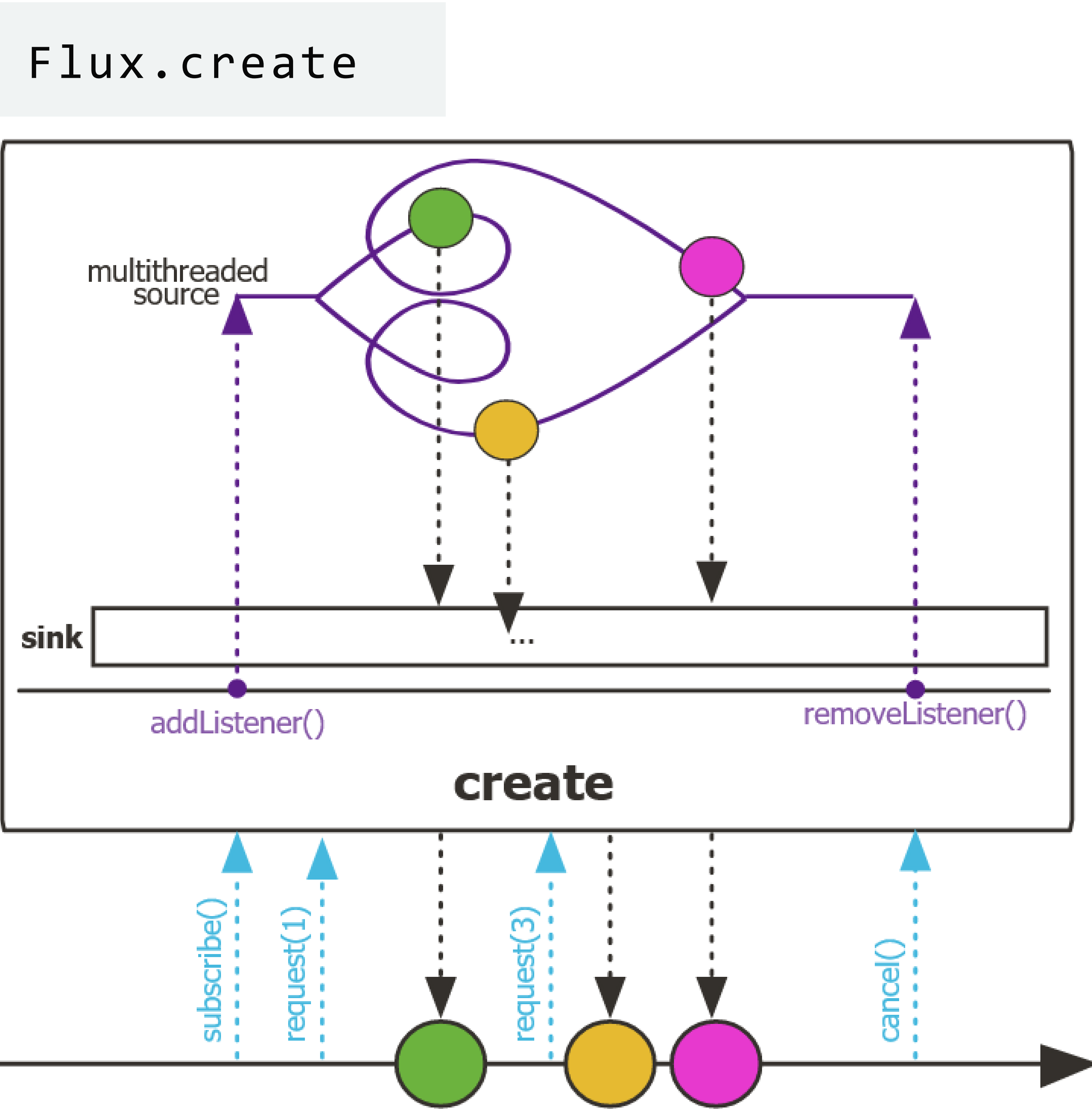
Mono - A Reactive Streams Publisher with basic rx operators that emits at most one item via the onNext signal then terminates with an onComplete signal (successful Mono, with or without value), or only emits a single onError signal (failed Mono).



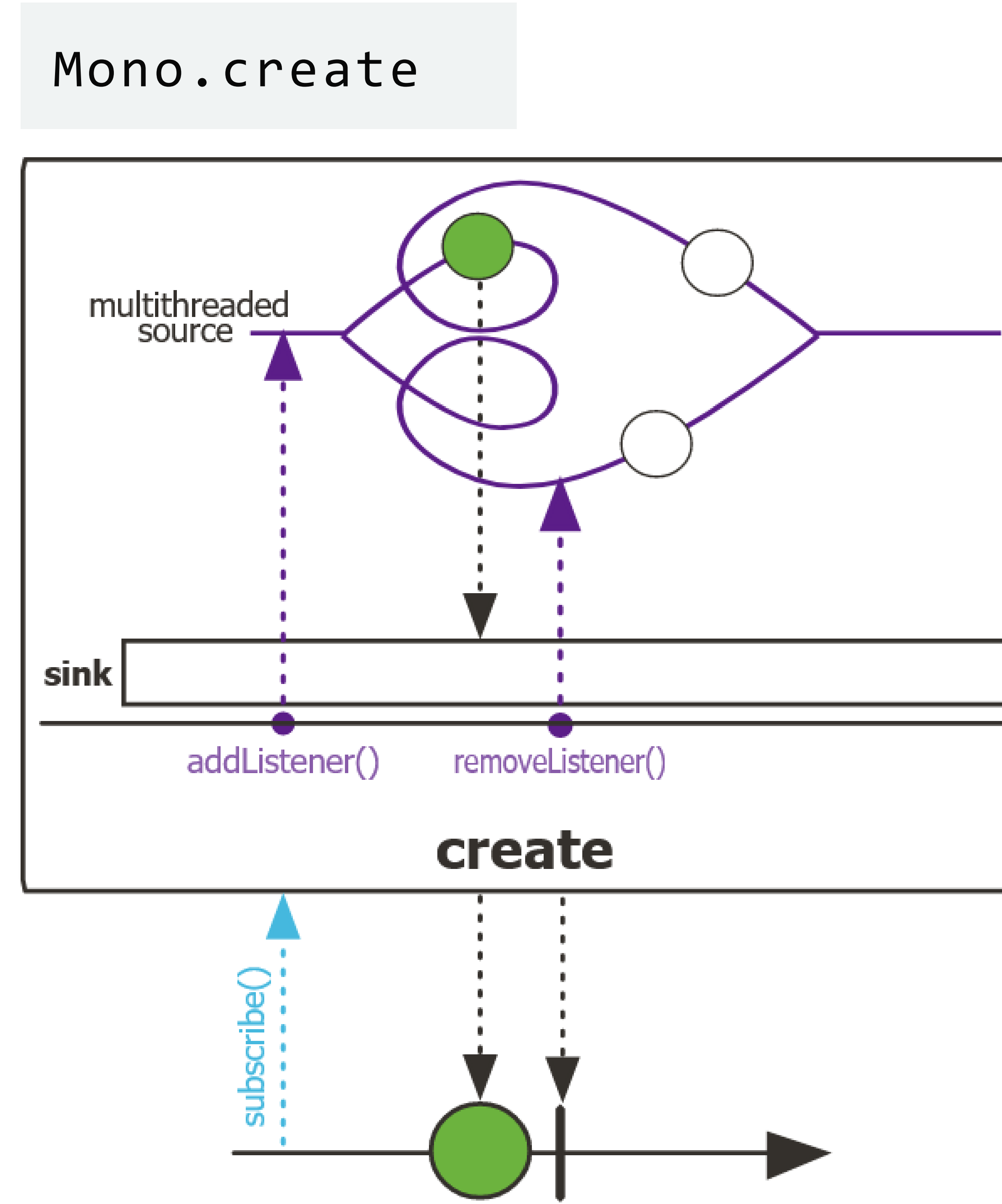
 © COPYRIGHT 2021 ENDAVA

Project Reactor Flux, Mono creation

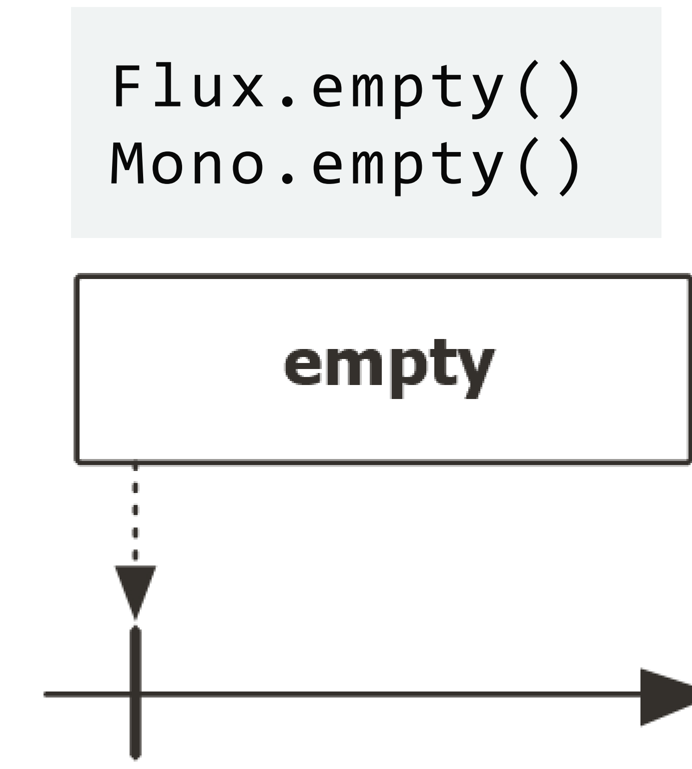
```
Flux.create
```




```
Mono.create
```




```
Flux.empty()
Mono.empty()
```



Project Reactor Demo (in Spring Boot)

Assembly Time vs Execution time

- Nothing happens until you subscribe
- Assembly time
- Execution time

```
Flux.range(1, 2)
    .map(i -> 10 + i)
    .map(i -> "value " + i)

    .subscribe()
```

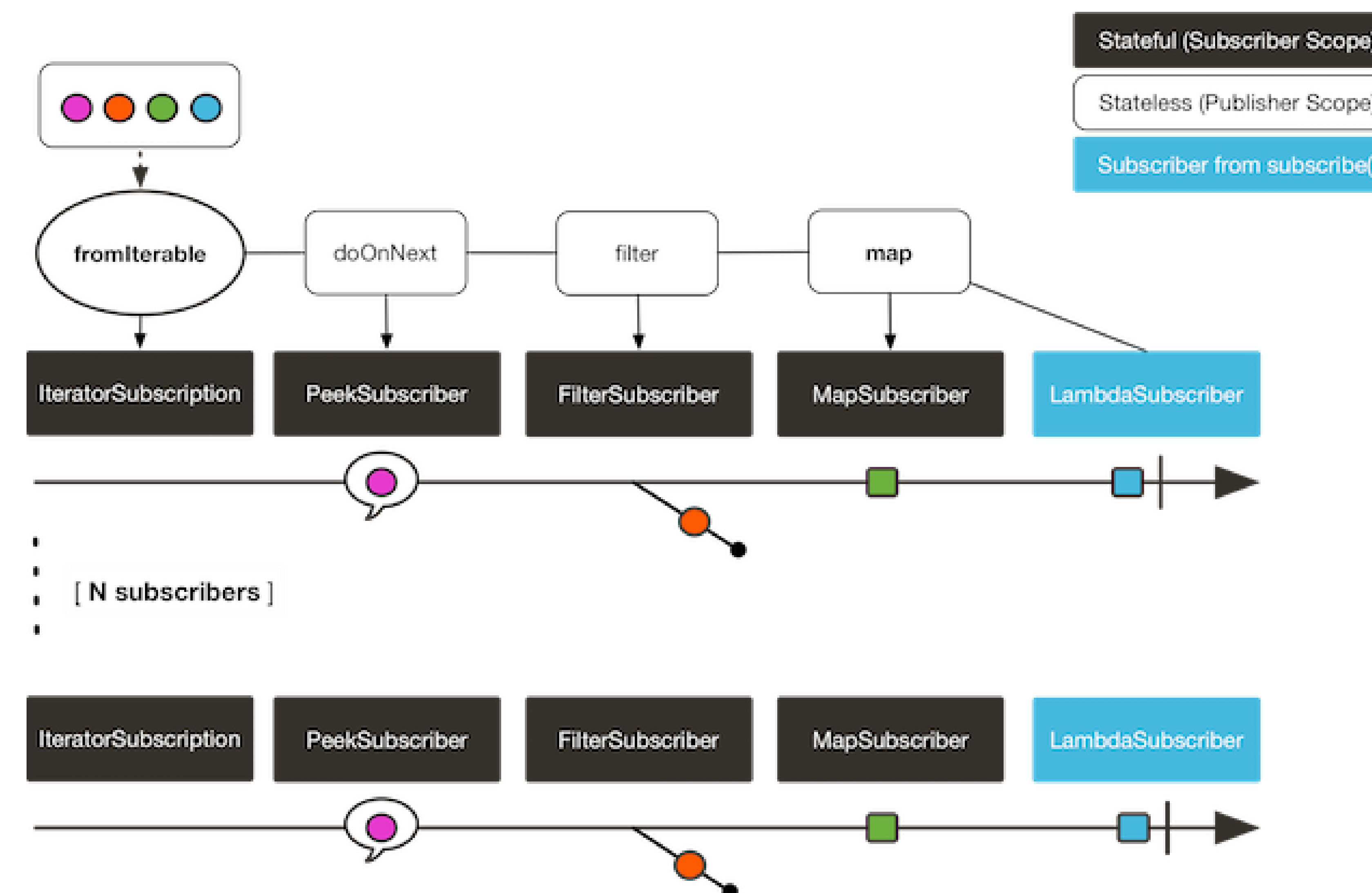
Cold and Hot Publisher

The Rx family of reactive libraries distinguishes two broad categories of reactive sequences: hot and cold. This distinction mainly has to do with how the reactive stream reacts to subscribers:

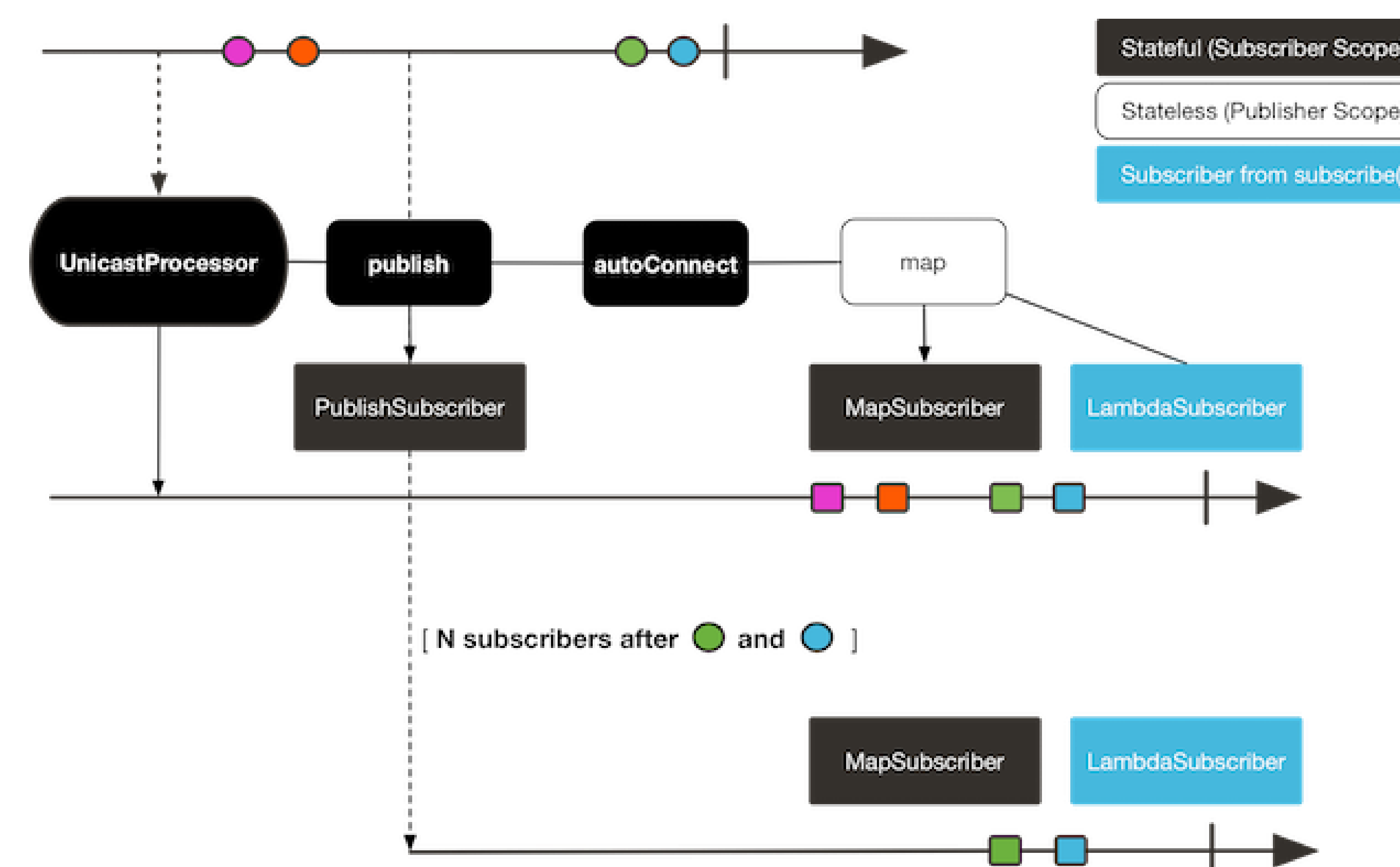
- A Cold sequence starts anew for each Subscriber, including at the source of data. For example, if the source wraps an HTTP call, a new HTTP request is made for each subscription.
- A Hot sequence does not start from scratch for each Subscriber. Rather, late subscribers receive signals emitted after they subscribed. Note, however, that some hot reactive streams can cache or replay the history of emissions totally or partially. From a general perspective, a hot sequence can even emit when no subscriber is listening (an exception to the "nothing happens before you subscribe" rule).

https://projectreactor.io/docs/core/release/reference/#reactor.hotCold

Cold Publisher



Hot Publisher



Schedulers

Reactor is concurrency-agnostic – execution model and where the execution happens is determined by currently used Scheduler

Scheduler types:

- Schedulers.immediate() – no execution context, submitted task is directly executed on the current thread
- Schedulers.single() – single, reusable thread
- Schedulers.boundedElastic() – worker pool suitable for I/O blocking work
- Schedulers.parallel() – fixed pool of workers tuned for parallel work (num of workers == CPU cores)

Using publishOn and subscribeOn operators

- subscribeOn applies to the subscription process, when that backward chain is constructed. As a consequence, no matter where you place the subscribeOn in the chain, it always affects the context of the source emission
- publishOn applies in the same way as any other operator, in the middle of the subscriber chain. It takes signals from upstream and replays them downstream while executing the callback on a worker from the associated Scheduler. Consequently, it affects where the subsequent operators execute

```
// wrapping synchronous, blocking call
Mono blockingWrapper = Mono.fromCallable(() -> {
    return /* make a remote synchronous call */
});
blockingWrapper = blockingWrapper.subscribeOn(Schedulers.boundedElastic());
```

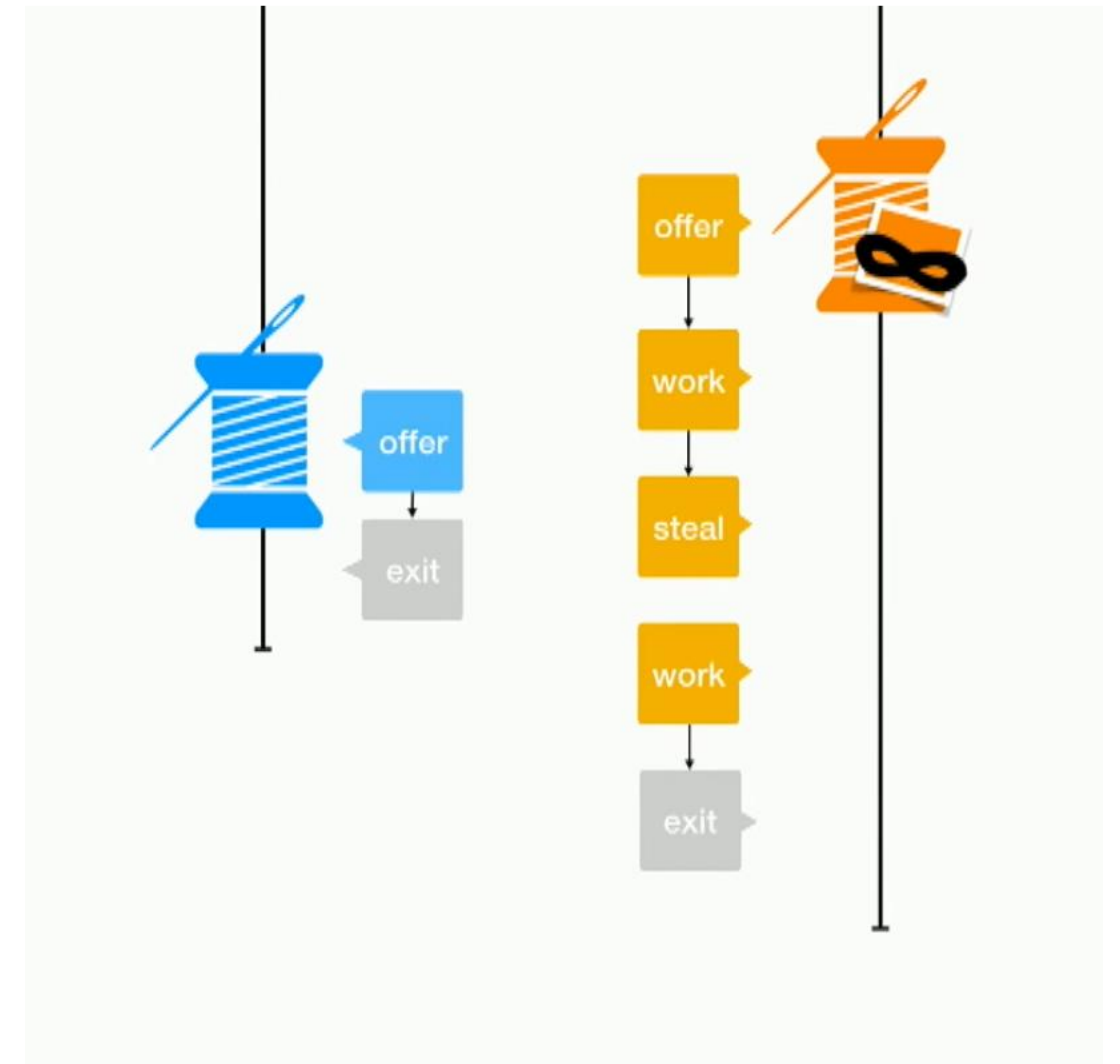
```
Flux<String> flux = Flux
    .range(1, 2)
    .map(i -> 10 + i) // caller thread
    .publishOn(Schedulers.parallel())
    .map(i -> "value " + i); // thread from Parallel Scheduler
```

```
Flux<String> flux = Flux
    .range(1, 2)
    .map(i -> 10 + i) // thread from Parallel Scheduler
    .subscribeOn(Schedulers.parallel())
    .map(i -> "value " + i); // thread from Parallel Scheduler
```

 © COPYRIGHT 2021 ENDAVA

Work Stealing

- One operator combining data from separate data sources (separate threads)
- Shared queue – thread sharing the queue offer data to it
- If there is already thread working, it will steal work offered from other threads



Source:https://youtu.be/hfupnixznp4

© COPYRIGHT 2021 ENDAVA

Operator Fusion

Support contract for queue-fusion based optimizations on subscriptions.

- Synchronous sources which have fixed size and can emit their items in a pull fashion, thus avoiding the request-accounting overhead in many cases.
- Asynchronous sources which can act as a queue and subscription at the same time, saving on allocating another queue most of the time.

```
Flux.range(1, 2)
    .map(i -> 10 + i)
    .map(i -> "value " + i)
```

 © COPYRIGHT 2021 ENDAVA

Q & A

endava