RXJava

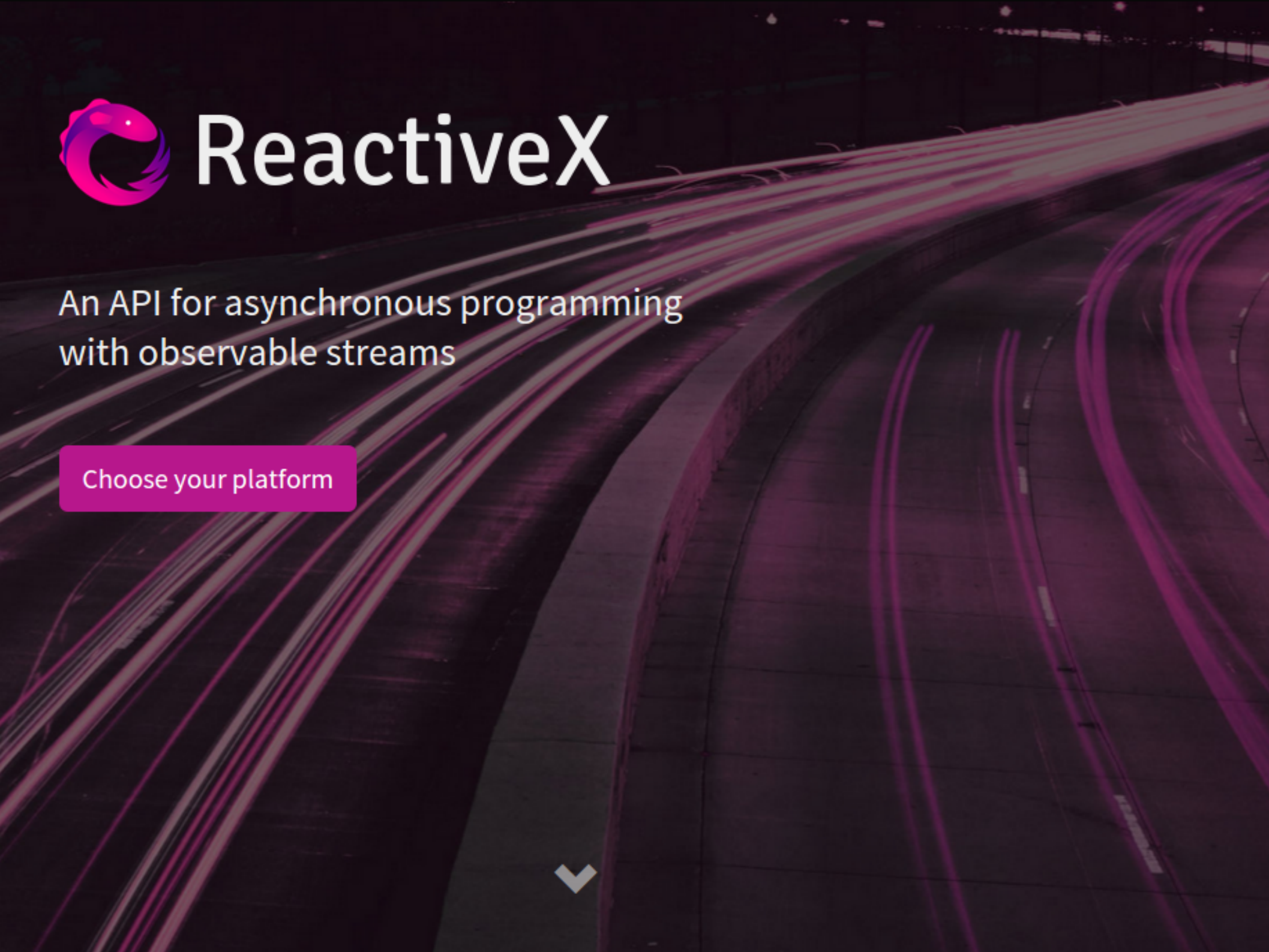
Daniel Hinojosa
@dhinojosa

About Me...

Programmer, Consultant, Trainer, Author:

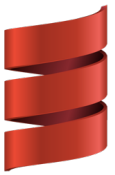
Testing in Scala (Book)
Beginning Scala Programming (Video)
Scala Beyond the Basics (O'Reilly Class)
TDD in Java (O'Reilly Class)

@dhinojosa
dhinojosa@evolutionnext.com


ReactiveX
An API for asynchronous programming
with observable streams
Choose your platform

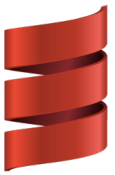
Reactive Extensions for Async Programming



RX
for Async Programming



RX Java

Definition

A library for composing asynchronous and event-based programs by using observable sequences.

Reactive Programming

Reacting to changes in data or events using functional abstractions

Reactive Programming

Responding to:
Mouse Clicks, Keyboard Events, Temperature Sensors, Stock Trades, Website Clicks, GPS Signals, HTTP Responses, MQ Messages, Kafka Messages, etc.

Unbiased

The source for any `Observable/Flowable` can from anywhere. Convert your arrays, futures, threads, into `Observables/Flowable`

formerly of....

Microsoft

"Mouse is a database"

Reactive Programming


NETFLIX
RXJava

- Push Model (Reactive)
- Pull Model (Interactive, Subscription Based)

Lazy

Synchronous/ Asynchronous

Synchronous by Default

Observer/
Iterator

Where can you use it?

Anywhere. It is a processing asynchronous glue in applications.

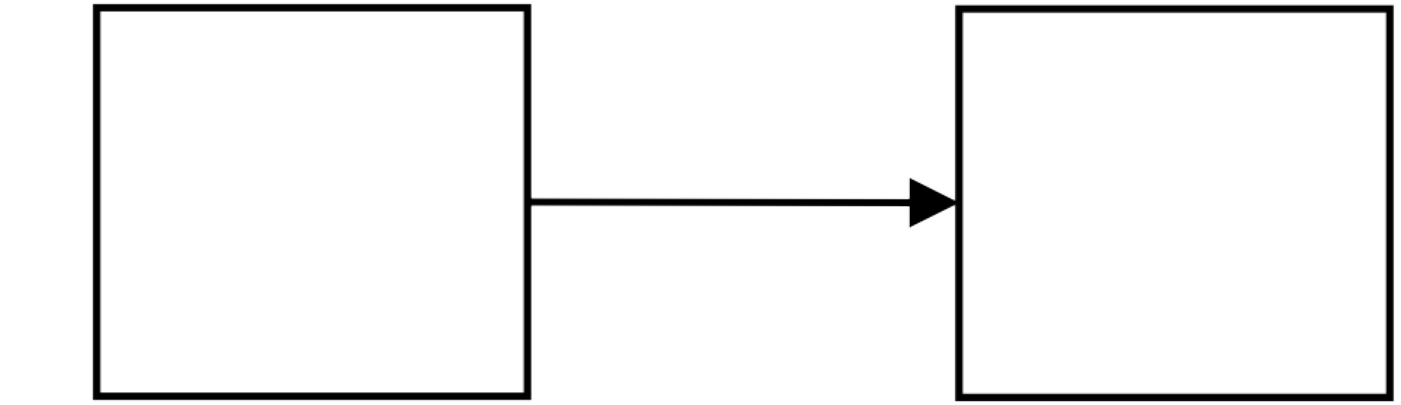
What pains does RXJava alleviate?

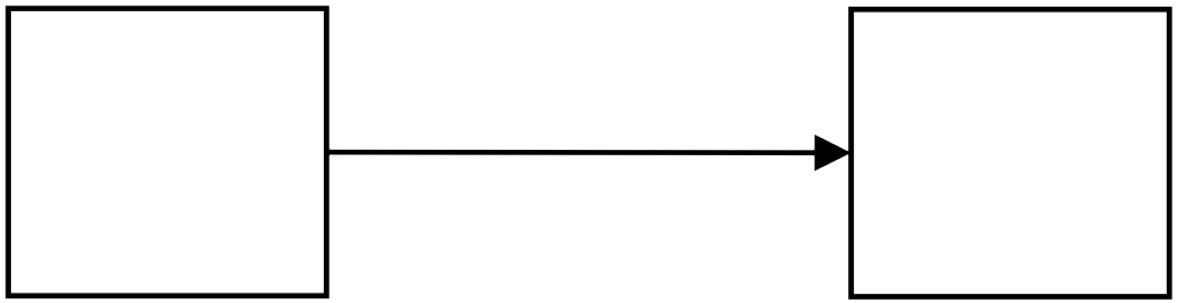
Callback Hell

Retaining a Source

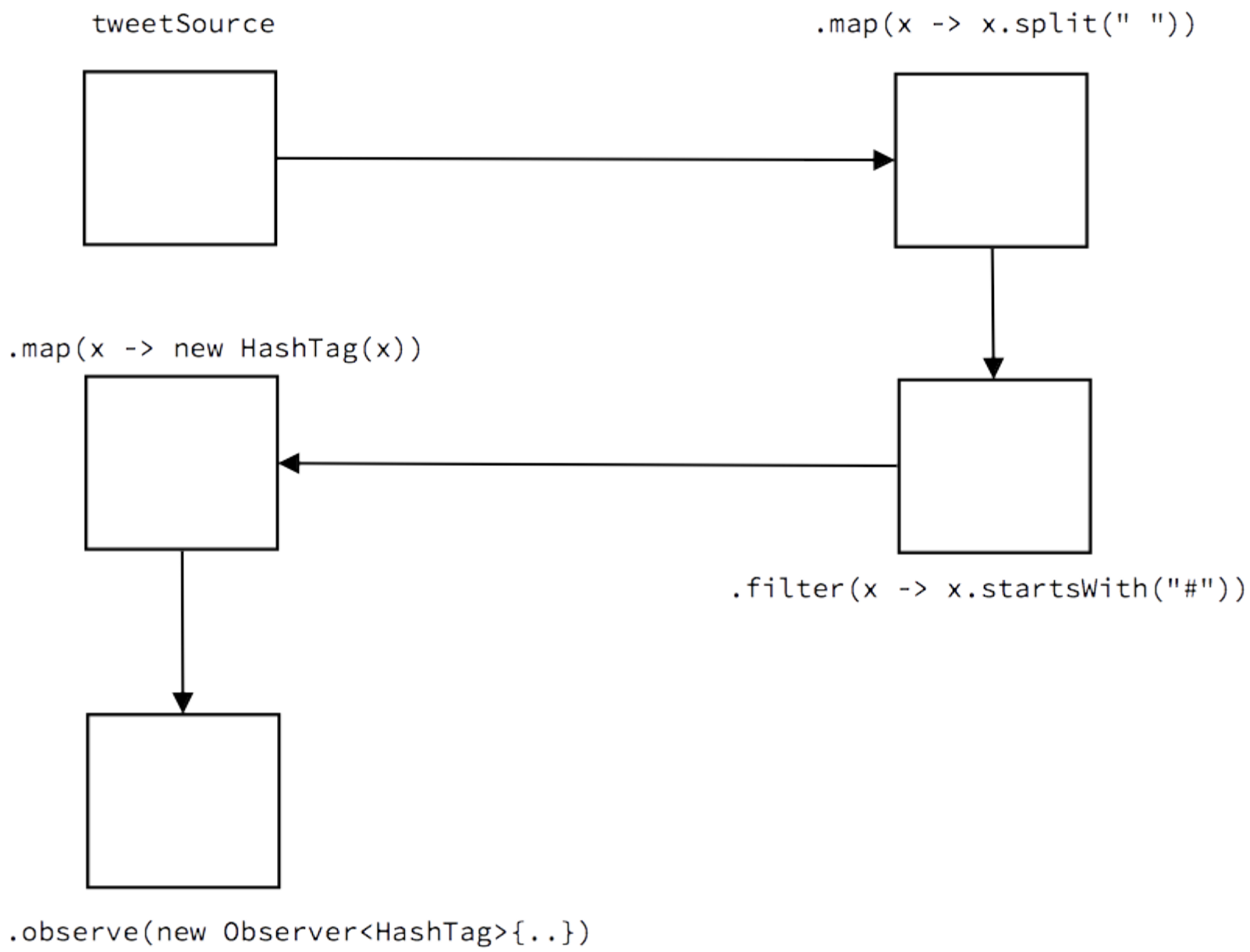
Backpressure Remedies

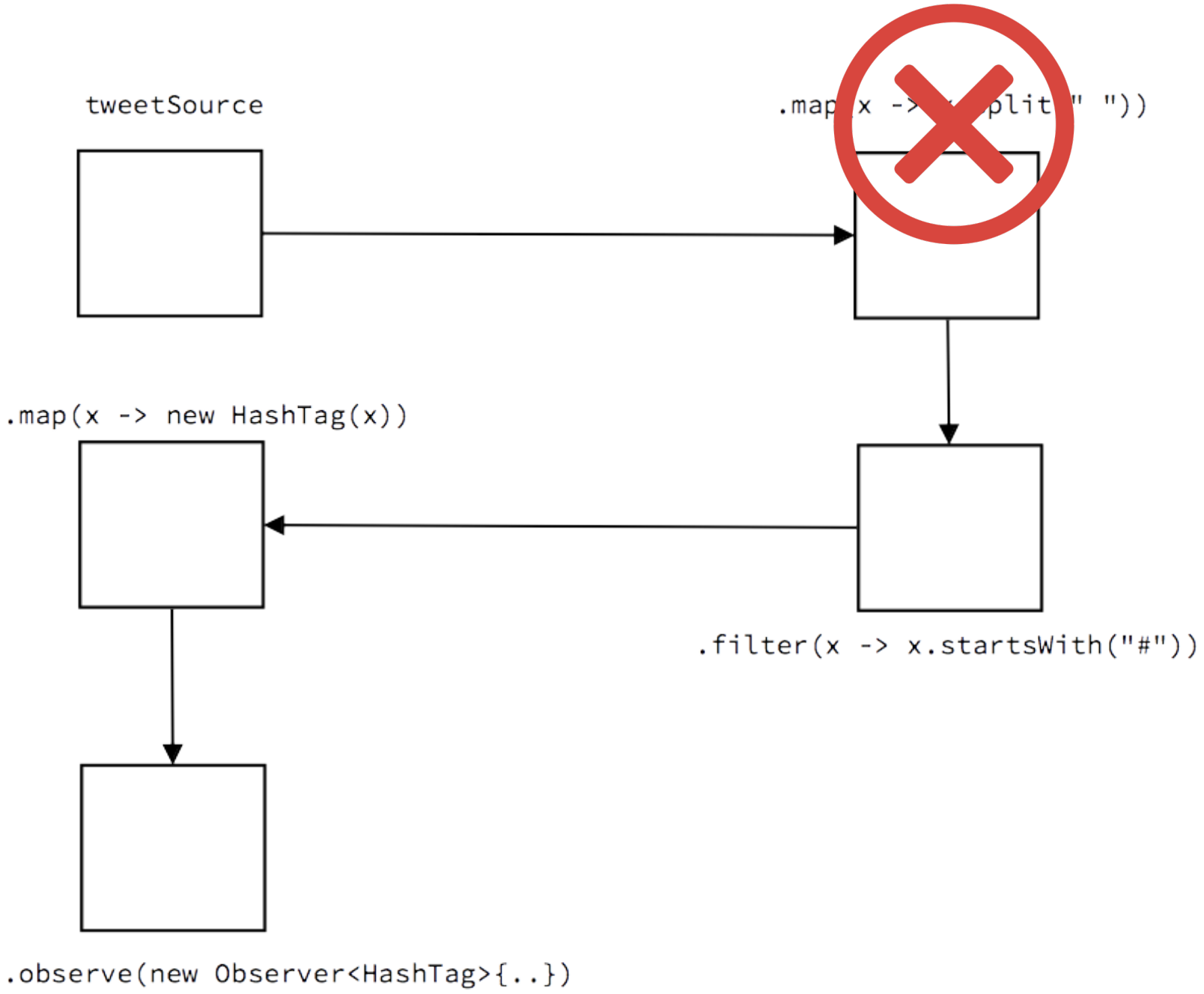
	One	Many
Sync	`T`	`Iterator<T>`
Asynchronous	`Future<T>`	`Observable<T>` `Flowable<T>`

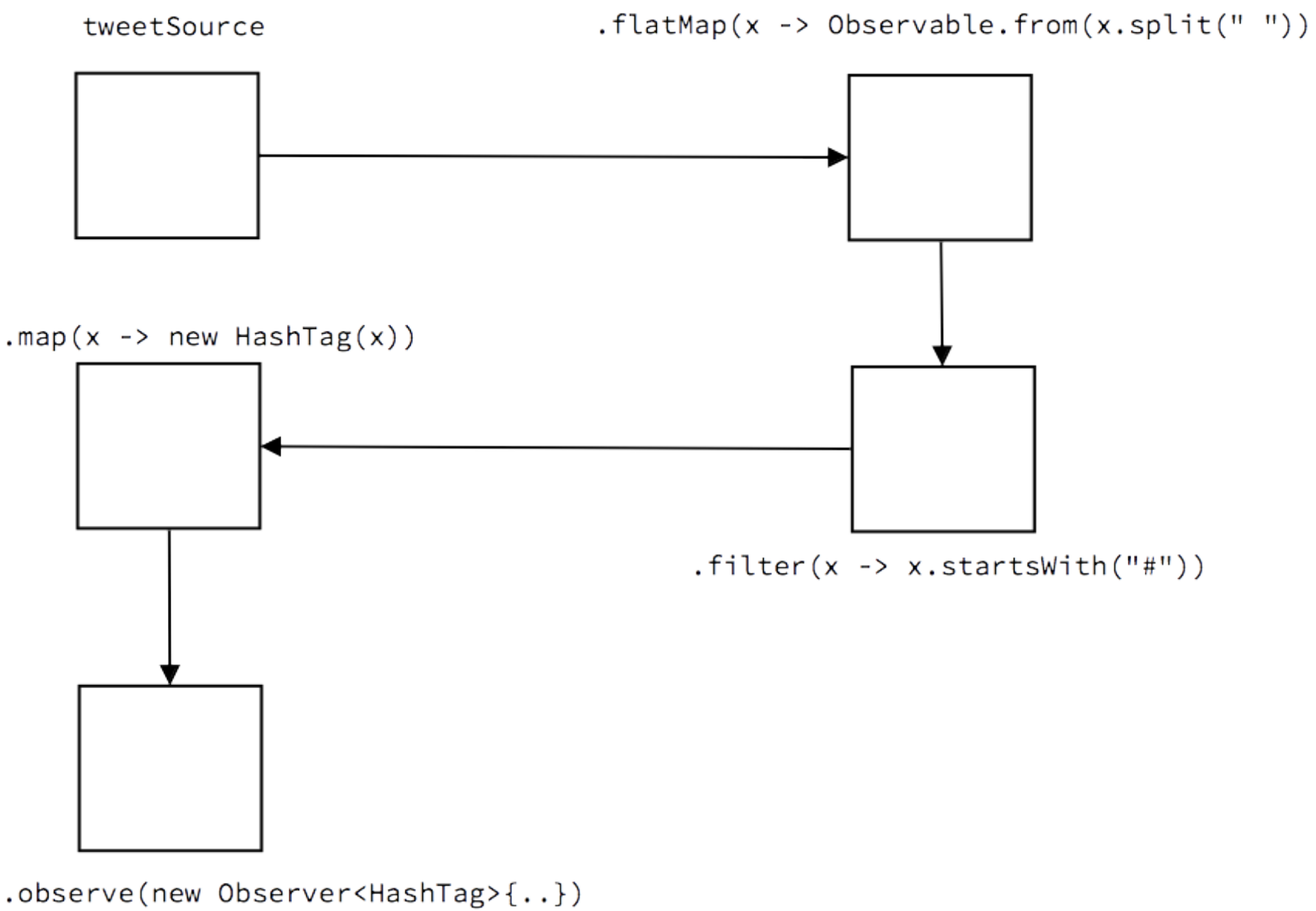

Observable<String>
Observer<String>

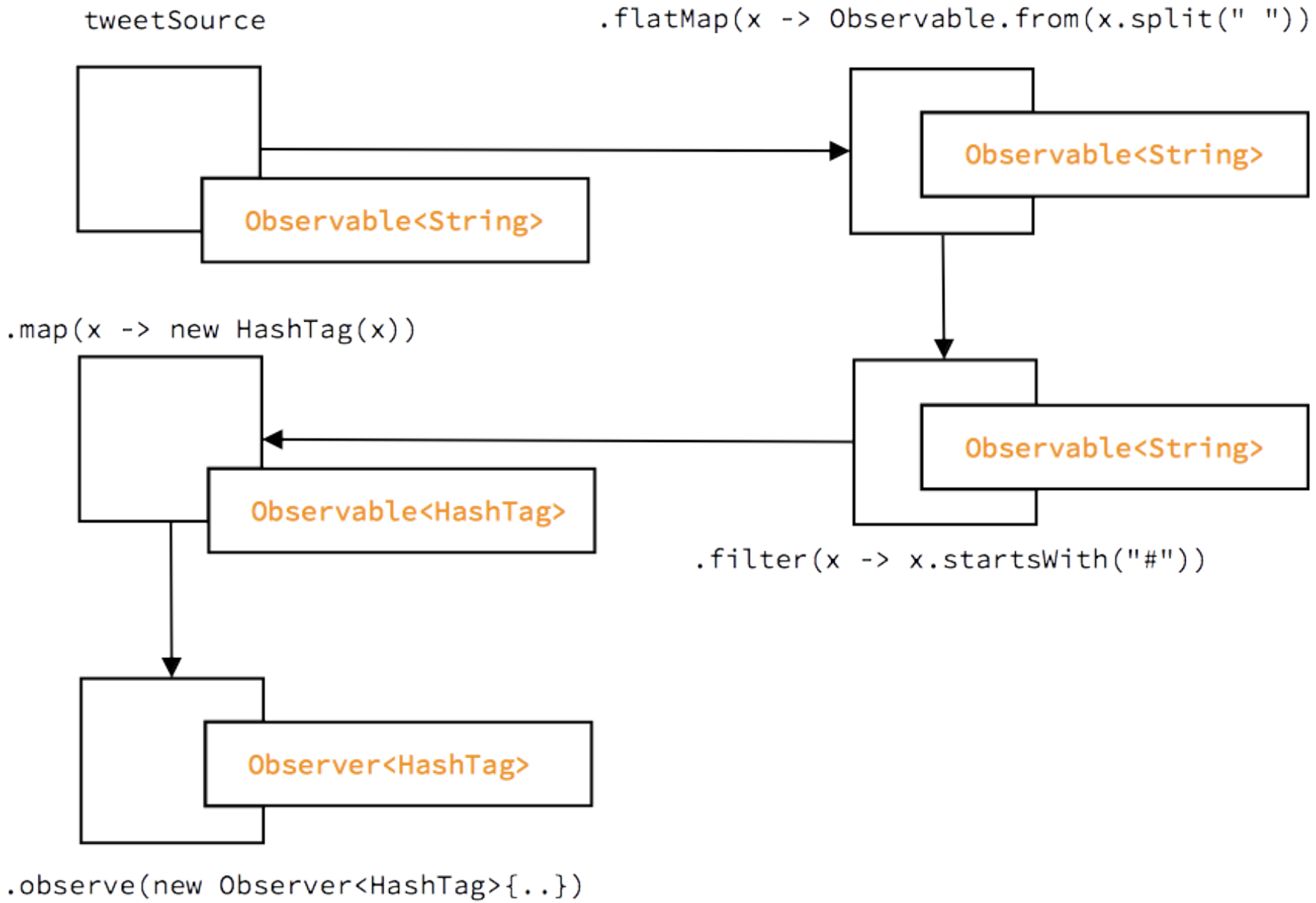


```
tweetSource        .observe(new Observer<String>() {...})
```


tweetSource
.map(x -> x.split(" "))
.map(x -> new HashTag(x))
.filter(x -> x.startsWith("#"))
.observe(new Observer<HashTag>{..})

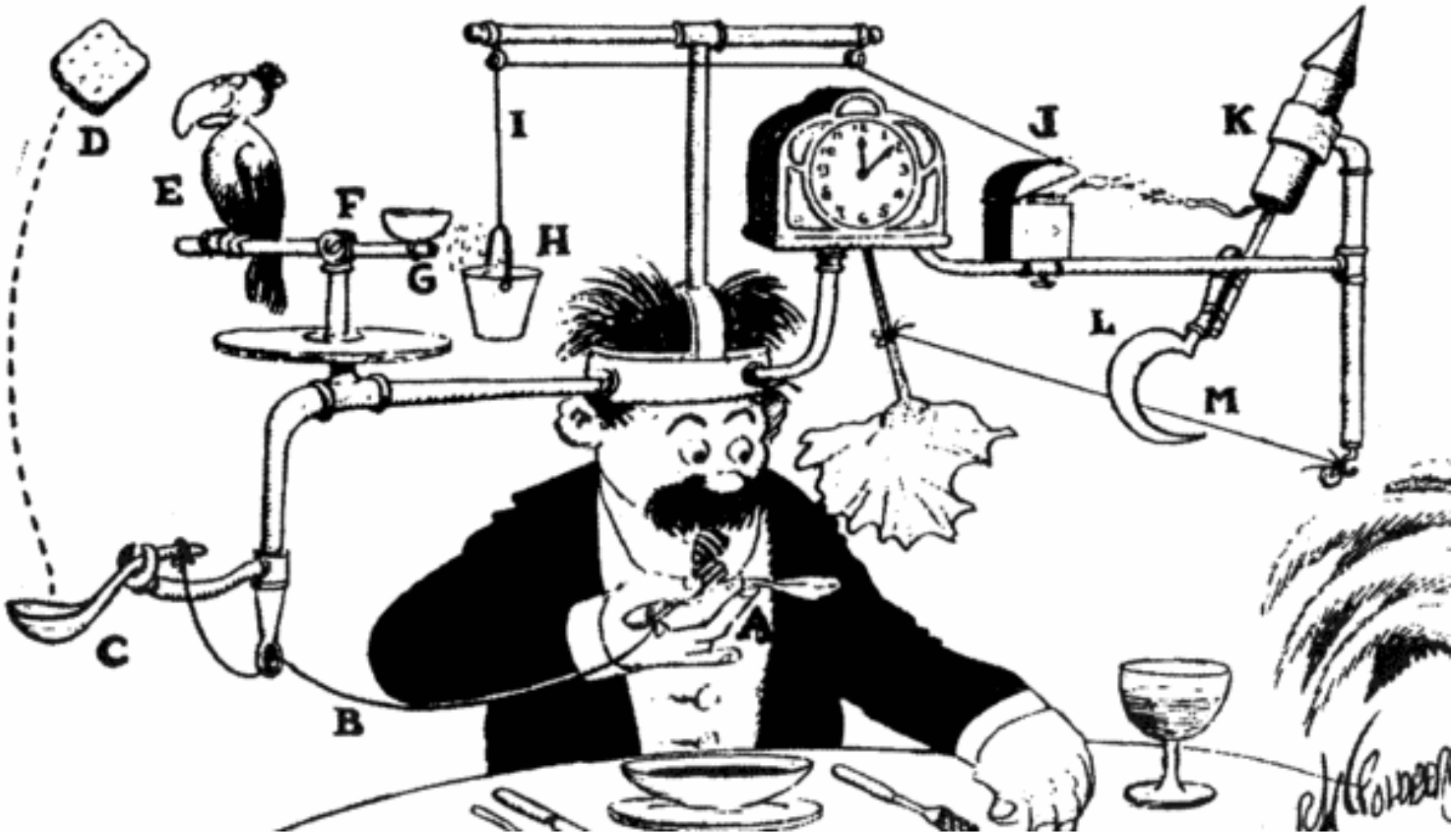

tweetSource
.map(x -> x.split(" "))
.map(x -> new HashTag(x))
.filter(x -> x.startsWith("#"))
.observe(new Observer<HashTag>{..})


tweetSource
.flatMap(x -> Observable.from(x.split(" "))
.map(x -> new HashTag(x))
.filter(x -> x.startsWith("#"))
.observe(new Observer<HashTag>{..})


tweetSource
Observable<String>
.flatMap(x -> Observable.from(x.split(" "))
Observable<String>
.map(x -> new HashTag(x))
Observable<HashTag>
Observable<String>
.filter(x -> x.startsWith("#"))
Observer<HashTag>
.observe(new Observer<HashTag>{..})

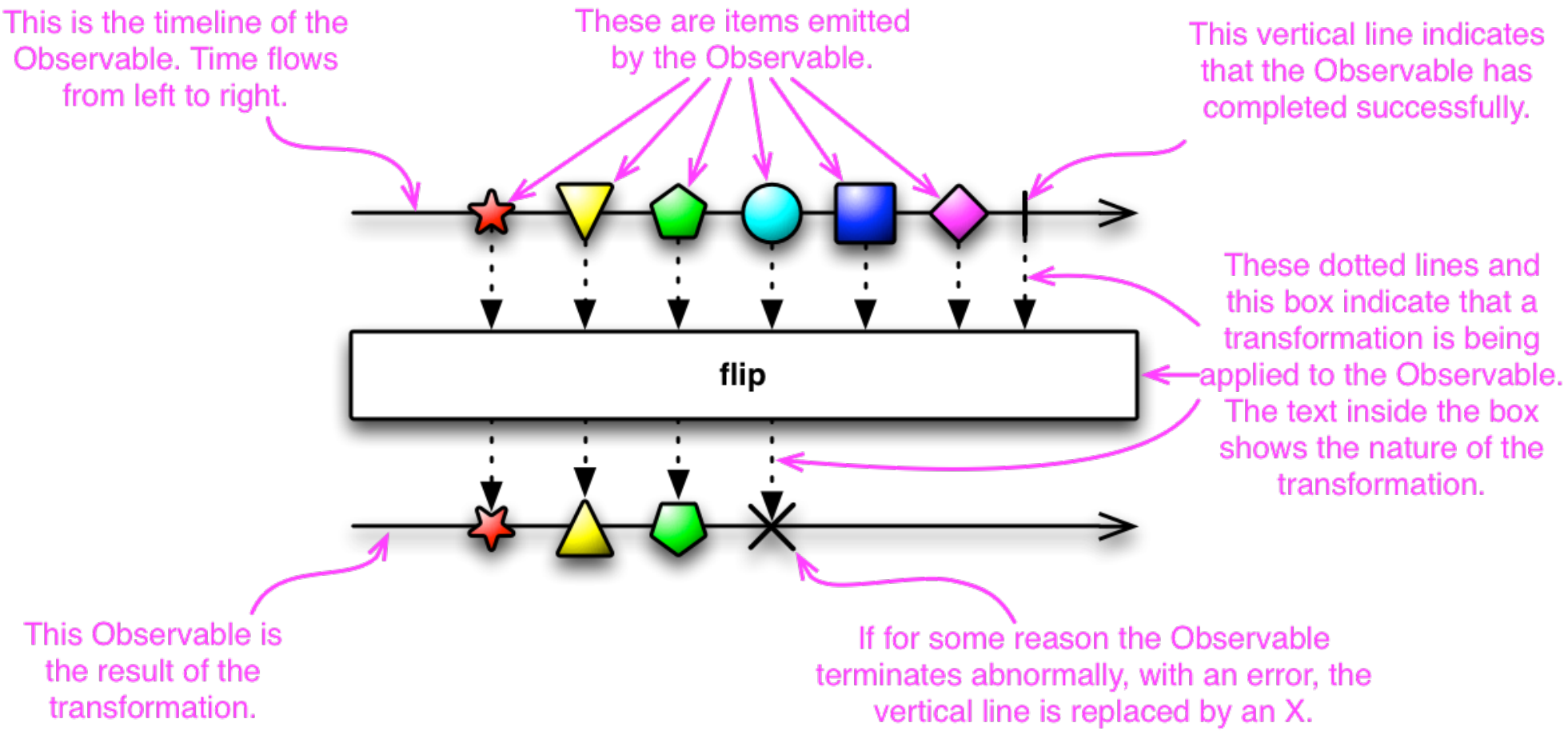
Thinking of `Observable`s as Pipes


Thinking of
Observables as
Lightning!



Thinking of `Observable`s as Rube Goldberg machines!

Marble Diagrams



Find all your favorite operators at:

http://reactivex.io/documentation/operators

Streams?

Reusing Streams

I'd like to duplicate a Java 8 stream so that I can deal with it twice. I can `collect` as a list and get new streams from that;

```
// doSomething() returns a stream
List<A> thing = doSomething().collect(toList());
thing.stream()... // do stuff
thing.stream()... // do other stuff
```

but I kind of think there should be a more efficient/elegant way. **Is there a way to copy the stream without turning it into a collection?**

Reusing Streams

38

I think your assumption about efficiency is kind of backwards. You get this huge efficiency payback if you're only going to use the data once, because you don't have to store it, and streams give you powerful "loop fusion" optimizations that let you flow the whole data efficiently through the pipeline.

If you want to re-use the same data, then by definition you either have to generate it twice (deterministically) or store it. If it already happens to be in a collection, great; then iterating it twice is cheap.

We did experiment in the design with "forked streams". What we found was that supporting this had real costs; it burdened the common case (use once) at the expense of the uncommon case. The big problem was dealing with "what happens when the two pipelines don't consume data at the same rate." Now you're back to buffering anyway. This was a feature that clearly didn't carry its weight.

If you want to operate on the same data repeatedly, either store it, or structure your operations as Consumers and do the following:

```
stream()...stuff....forEach(e -> { consumerA(e); consumerB(e); });
```

You might also look into the RxJava library, as its processing model lends itself better to this kind of "stream forking".

share edit flag

edited May 26 '14 at 4:44

answered May 26 '14 at 0:00

Brian Goetz

25.5k ● 7 ● 53 ● 71

What's the difference between 1.x and 2.x and 3.x?

Different Typed Streams

`Observables` is non-backpressured.
New type `Flowable` is backpressured.
New `Single` type for emitting a single item
New `Maybe` type for 0 or 1 items
New `Completable` for no return types

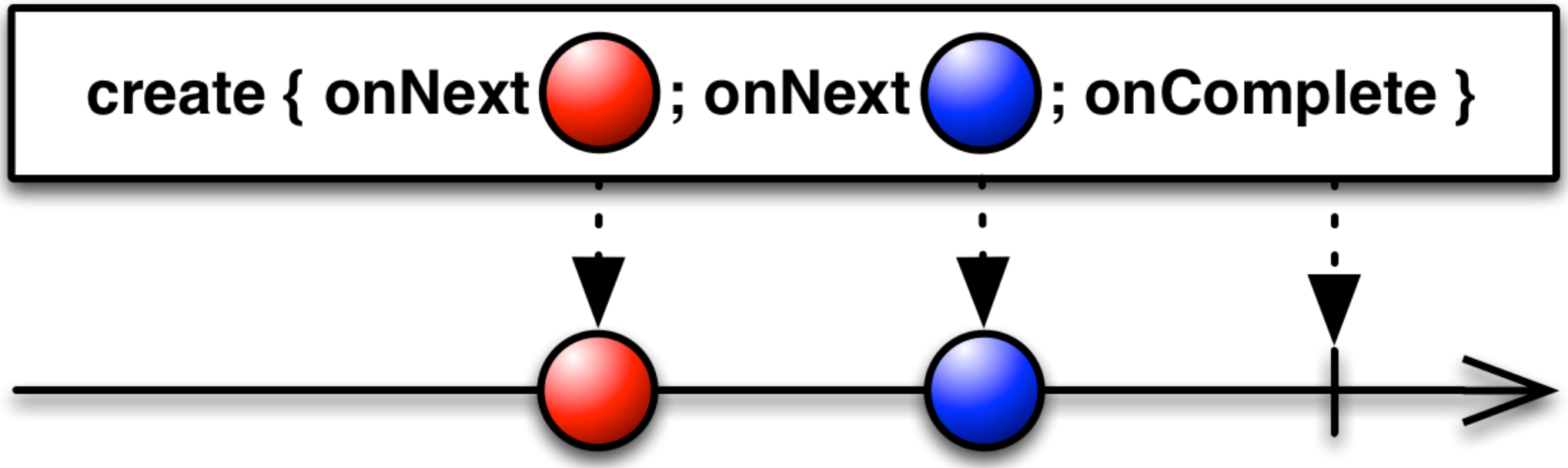
RXJava 3

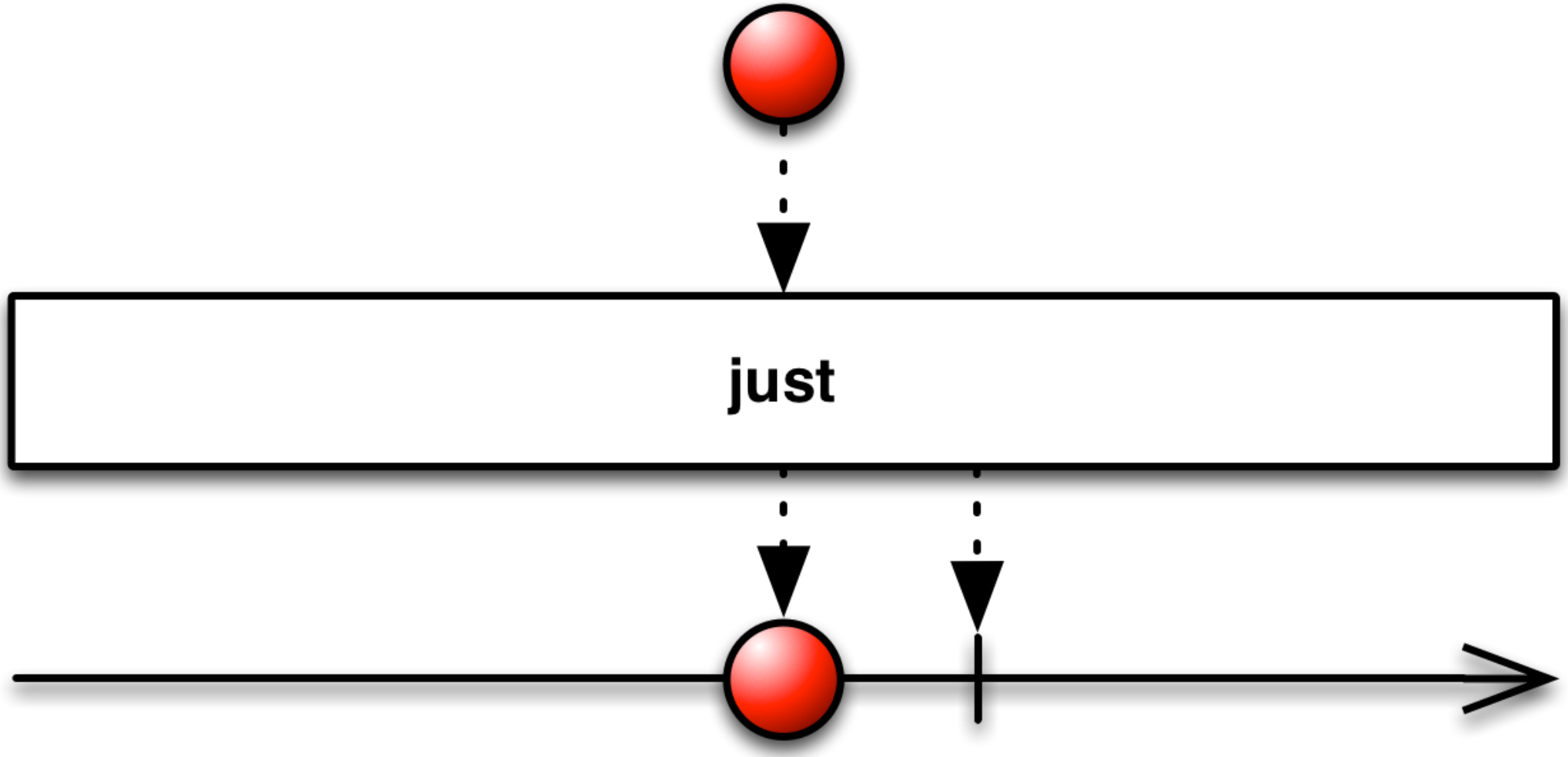
- All Functions are now Java 8 Standard
- Applies to reactive-streams.org and J9 Standards
- https://github.com/ReactiveX/RxJava/wiki/What's-different-in-3.0

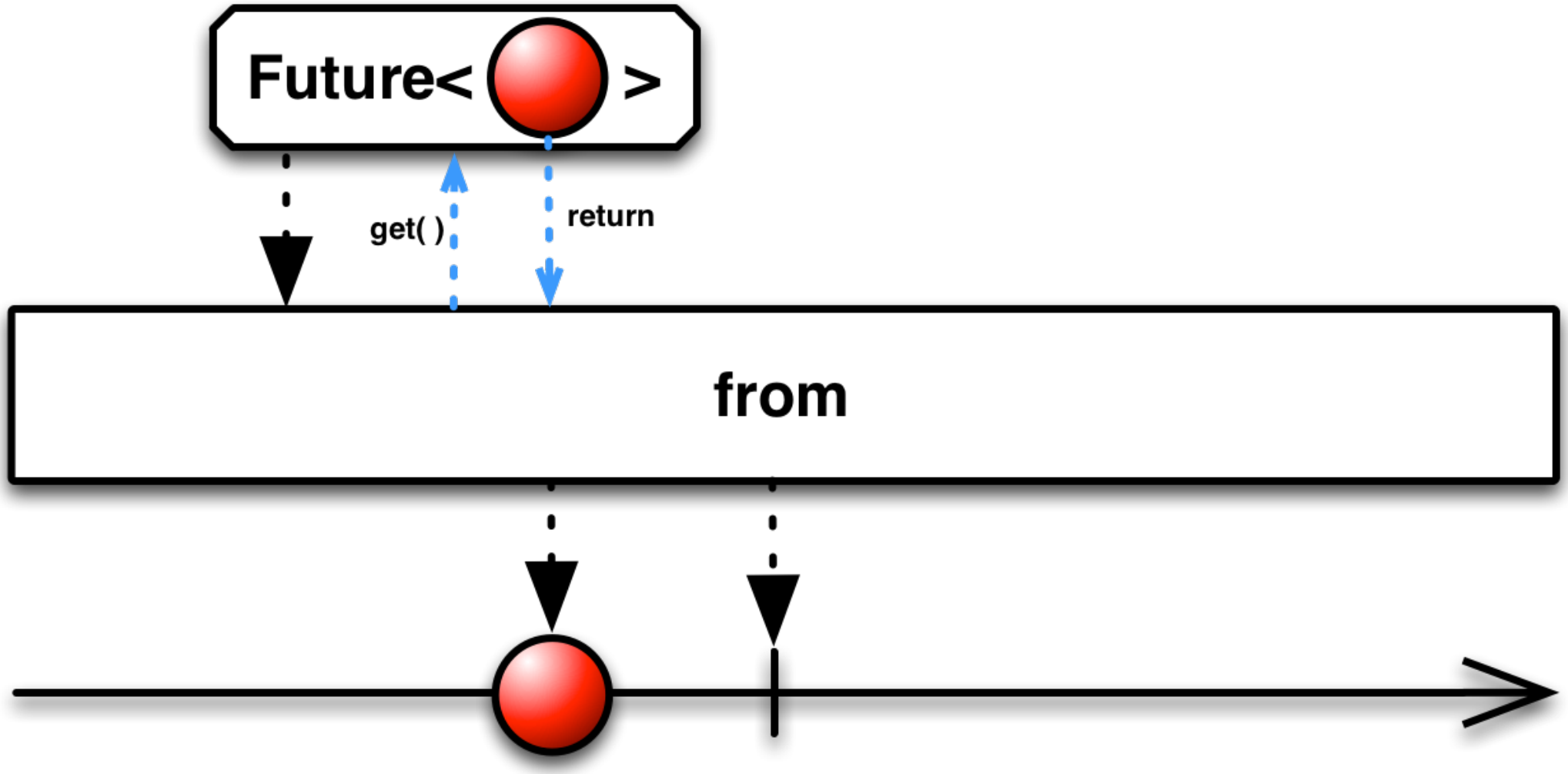
Lab: Java 9 Reactive Streams

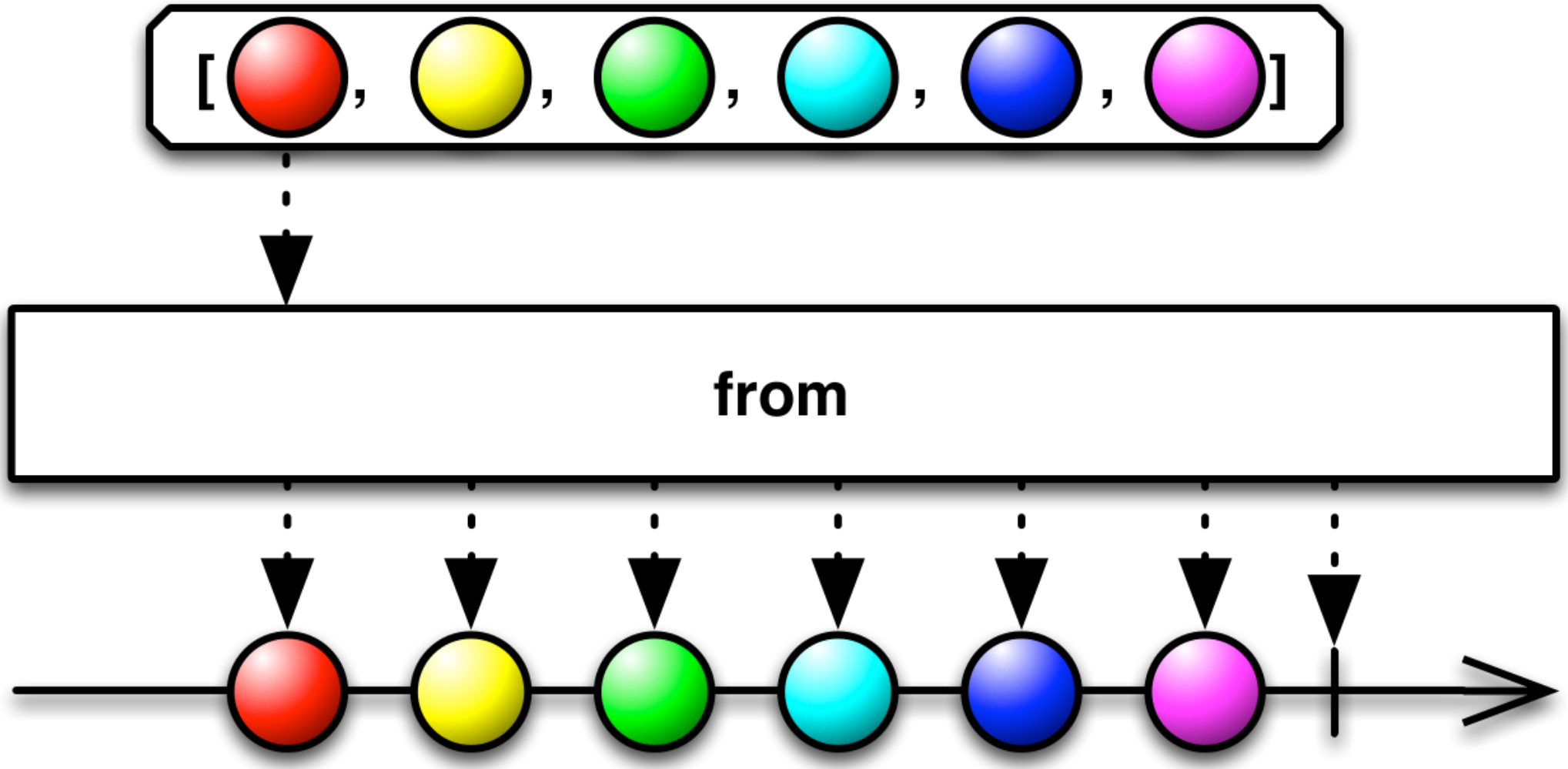
	None	One	Many
Sync	`void doAction()`	`T`	`Iterator<T>`
Asynchronous	`Completable<T>`	`Single<T>`	`Observable<T>` `Flowable<T>`

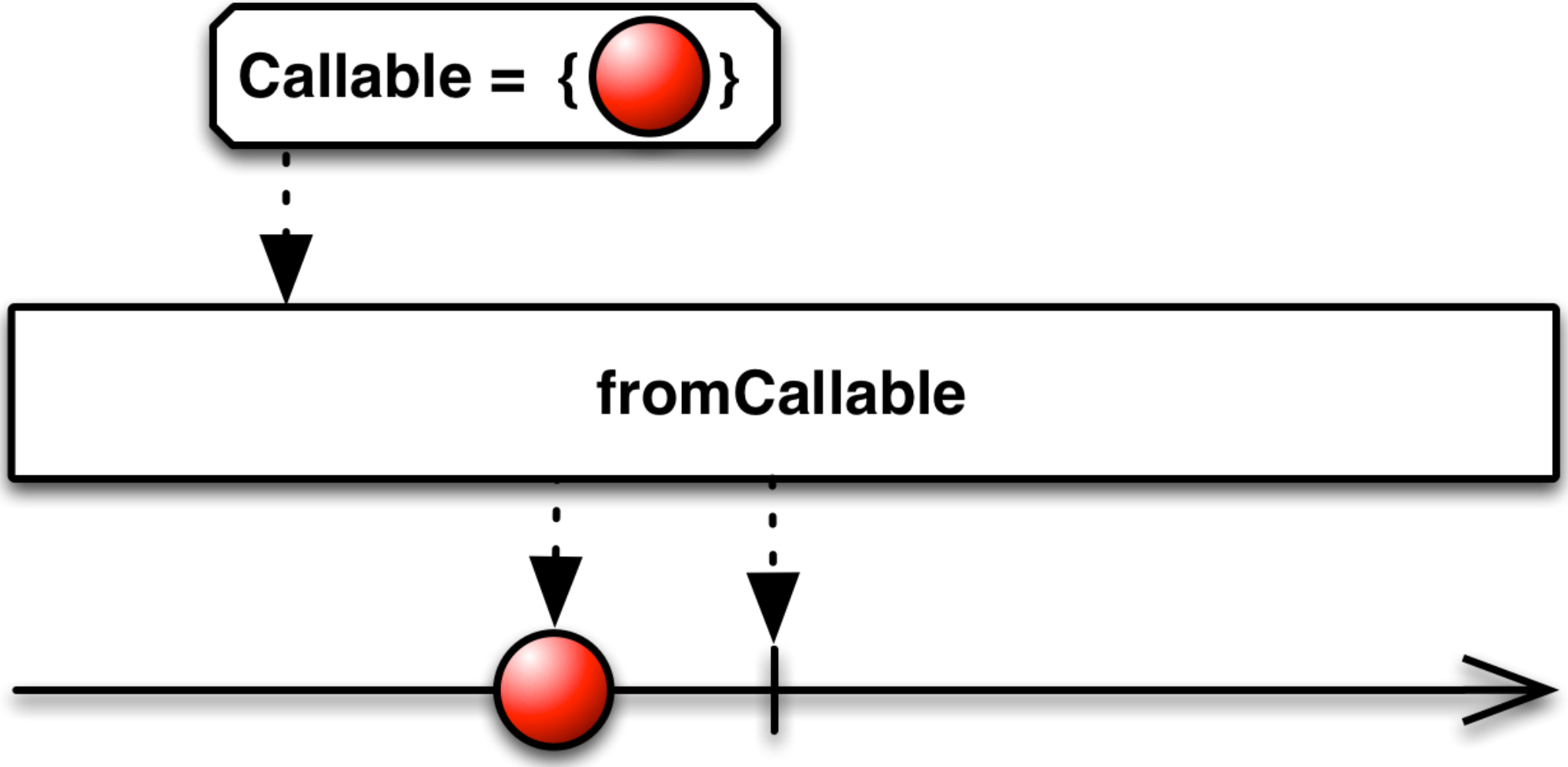
Creation of Observables & Flowables

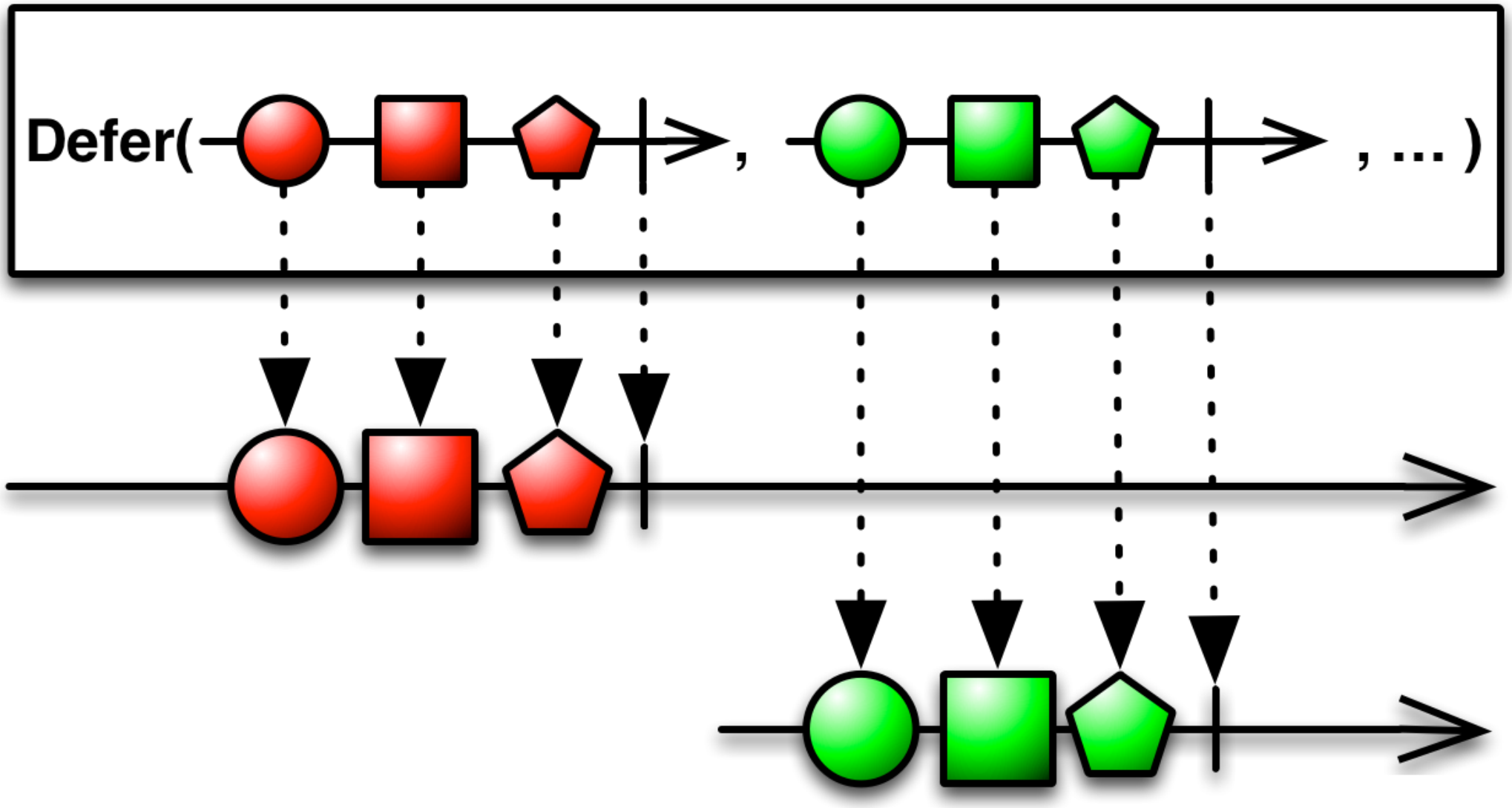

create { onNext ; onNext ; onComplete }

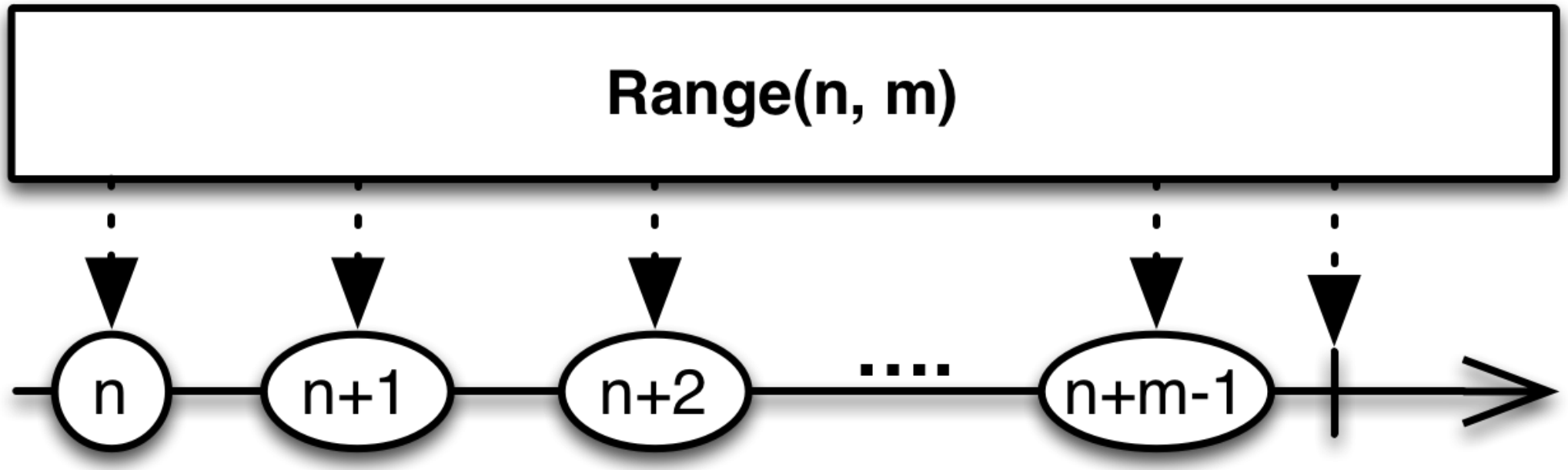

just

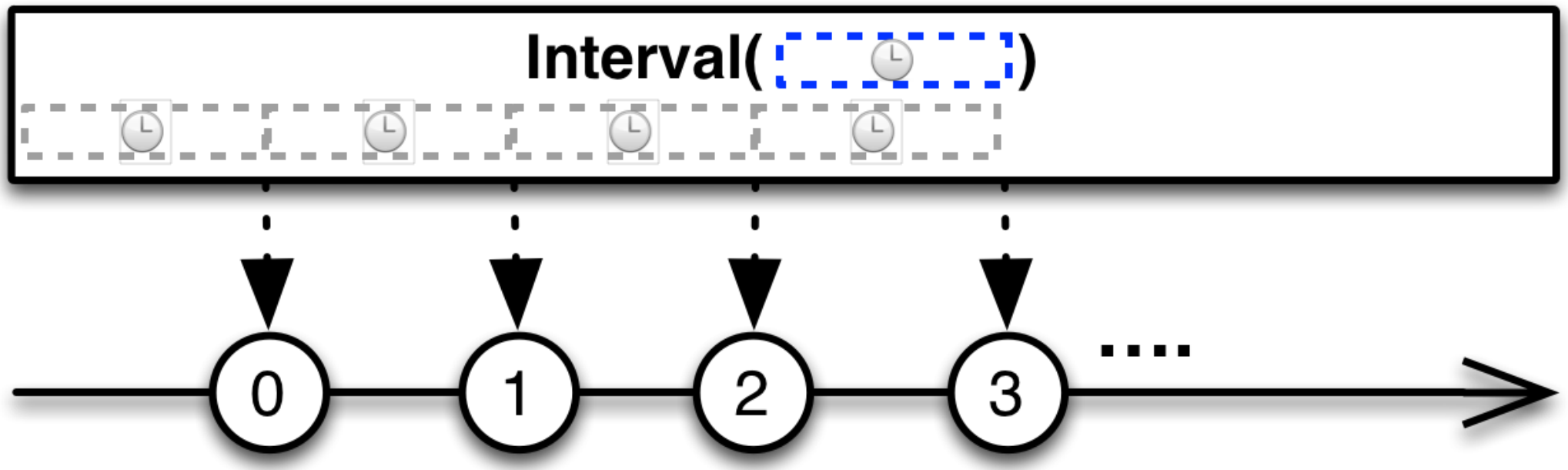

Future< >
get( )
return
from


[, , , , ,]
from


Callable = {}
fromCallable


Defer(
,
, ...)


Range(n, m)
n
n+1
n+2
....
n+m-1


Interval( )
0
1
2
3
....

Creation of Observables

Observables can be created from an `Iterables`, `Futures`, or `Functions`.

The RXJava Contract

The RXJava Contract

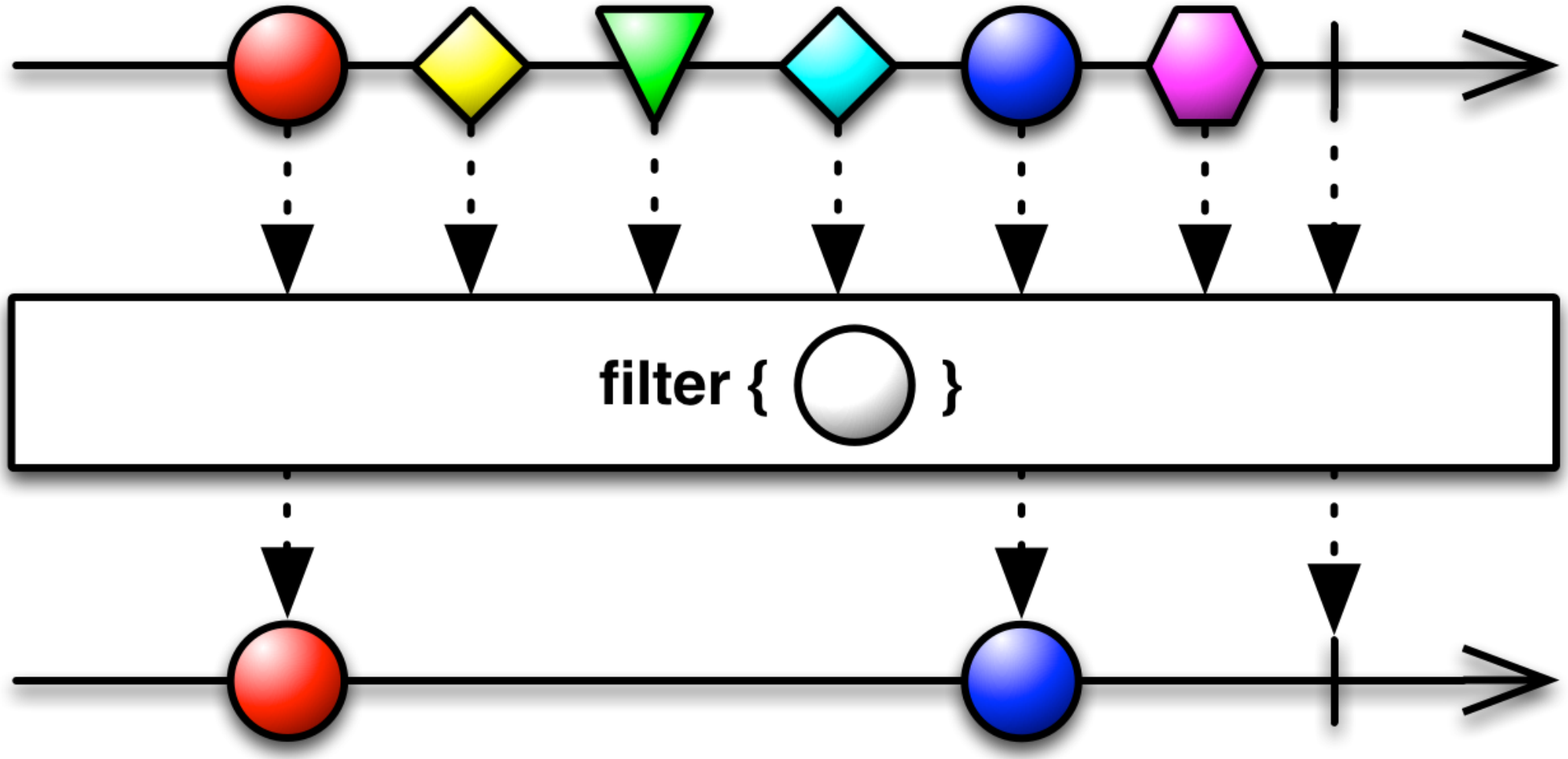
The events (`onNext()`, `onCompleted()`, `onError()`) can never be emitted concurrently

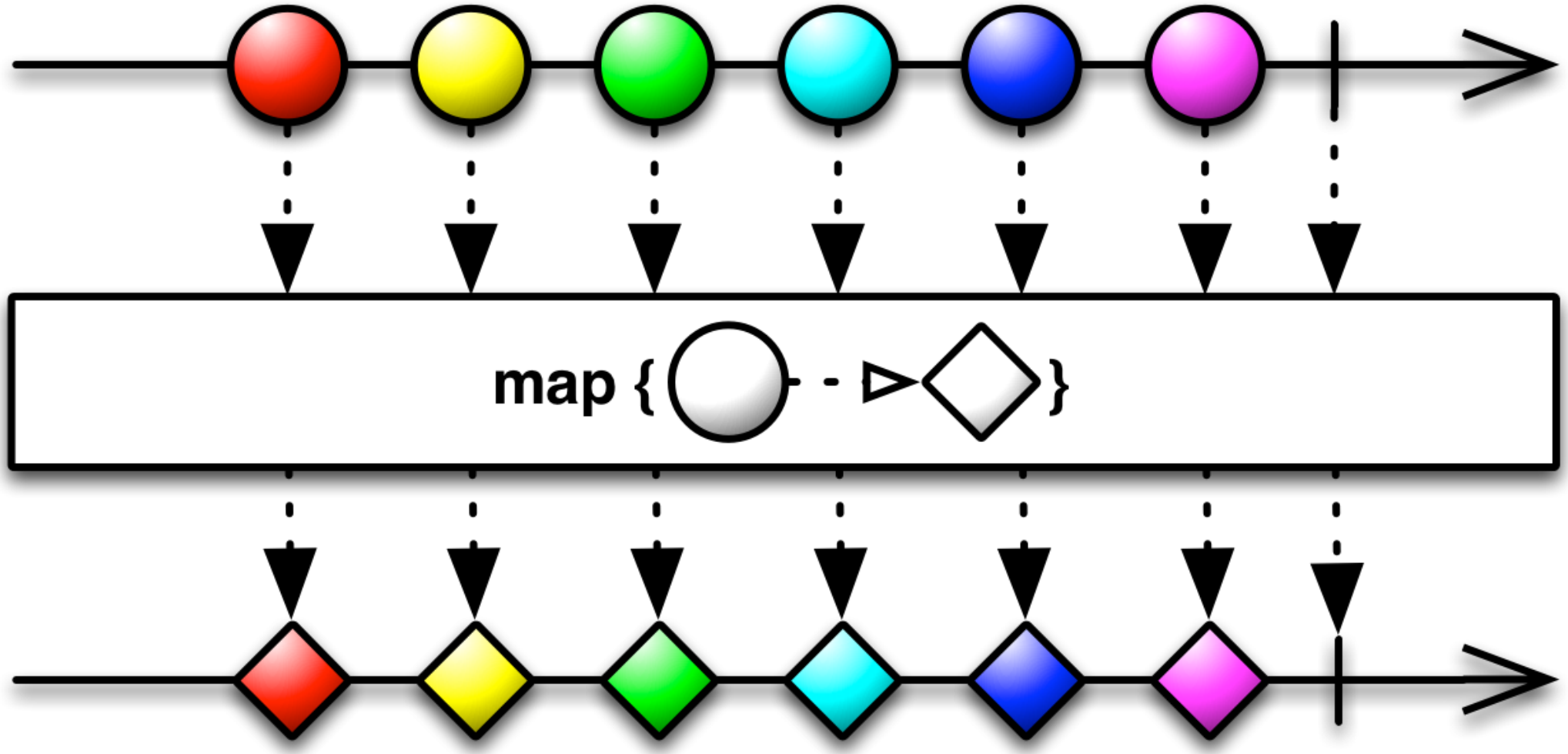
The RXJava Contract

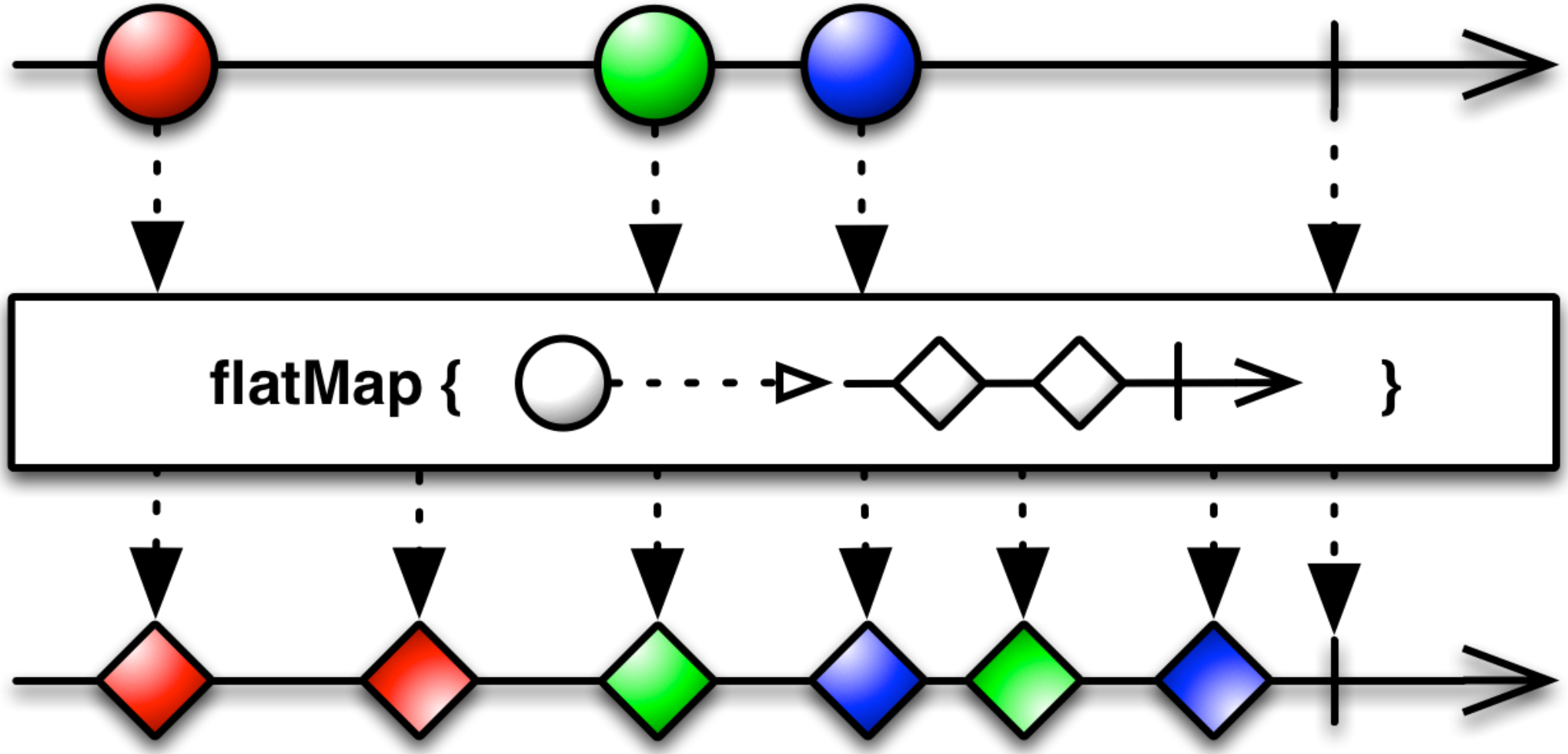
```
Observable.create(s -> {
  // Thread A
  new Thread(() -> {
    s.onNext("one");
    s.onNext("two");
  }).start();

  // Thread B
  new Thread(() -> {
    s.onNext("three");
    s.onNext("four");
  }).start();
```

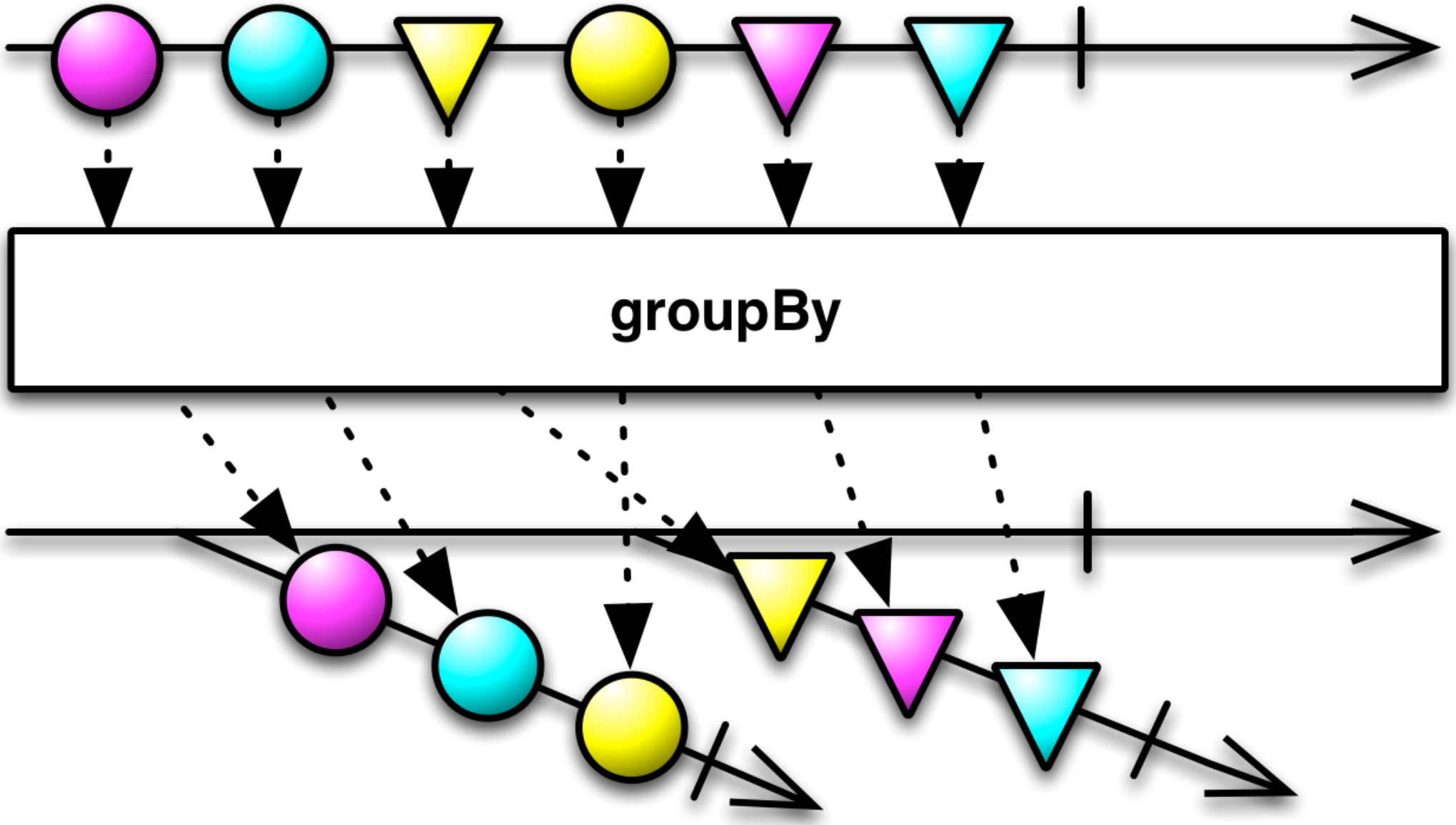
Lab: Creation of Observables

Standard Functional Operators


filter { }

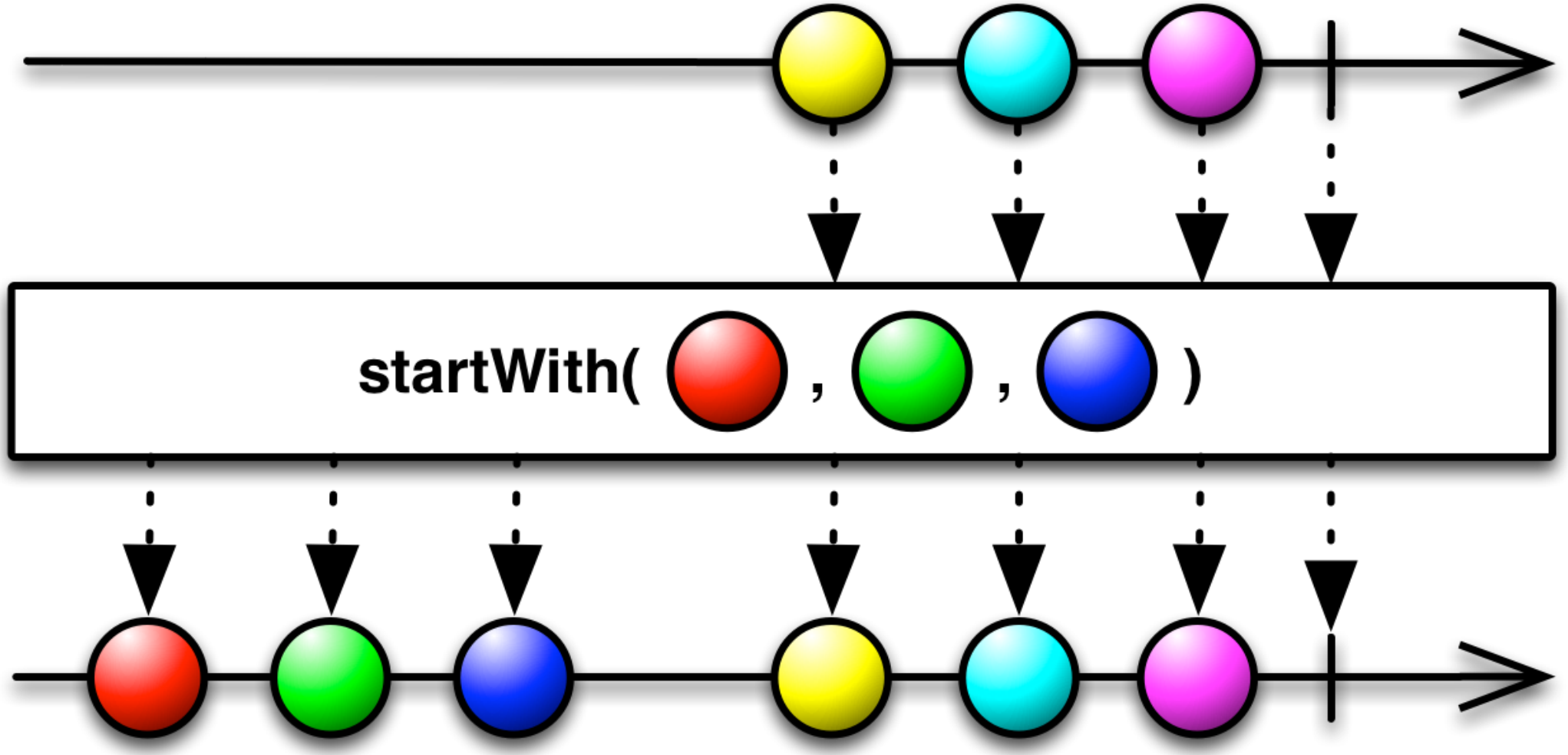

map {

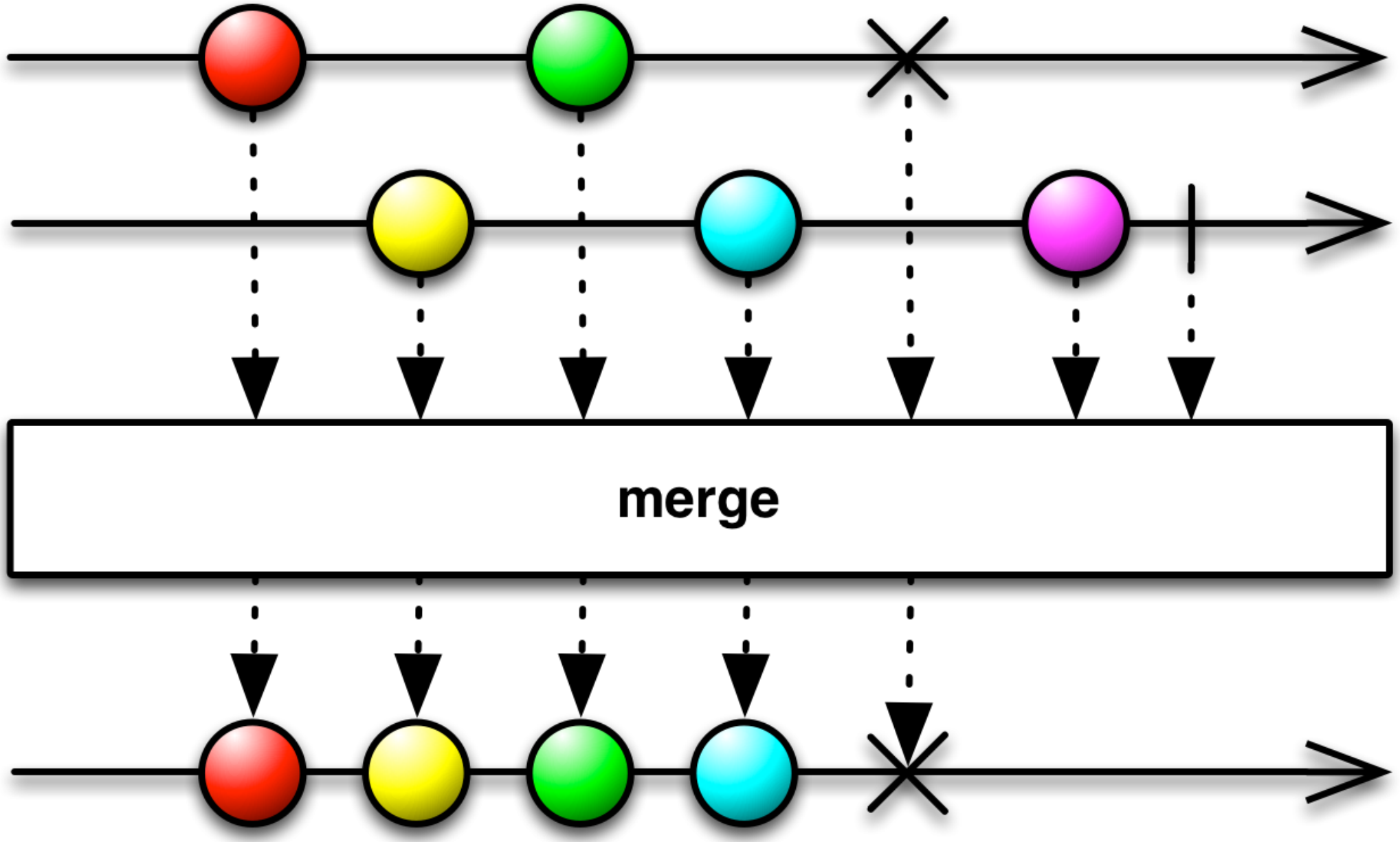

flatMap {
}

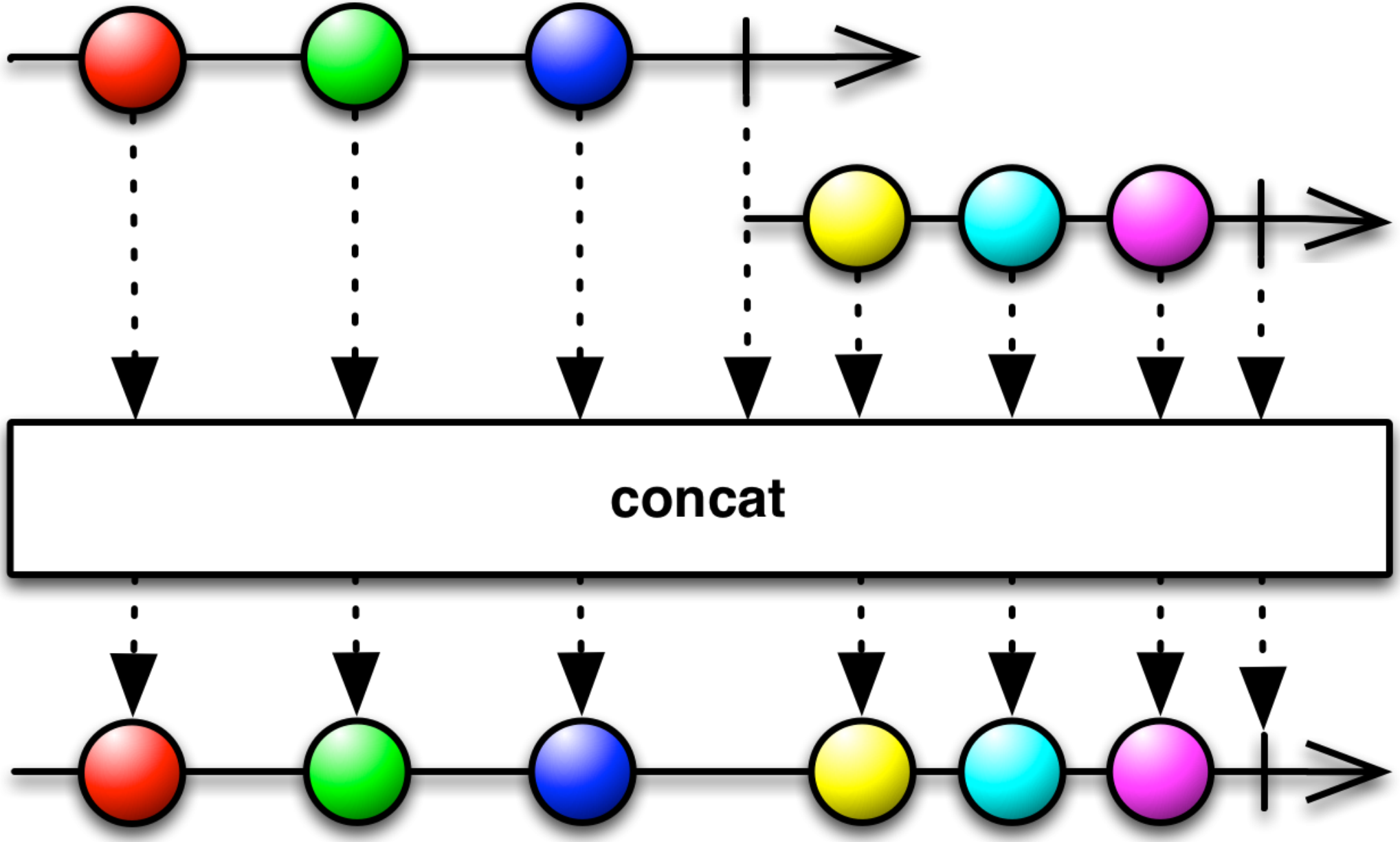

groupBy

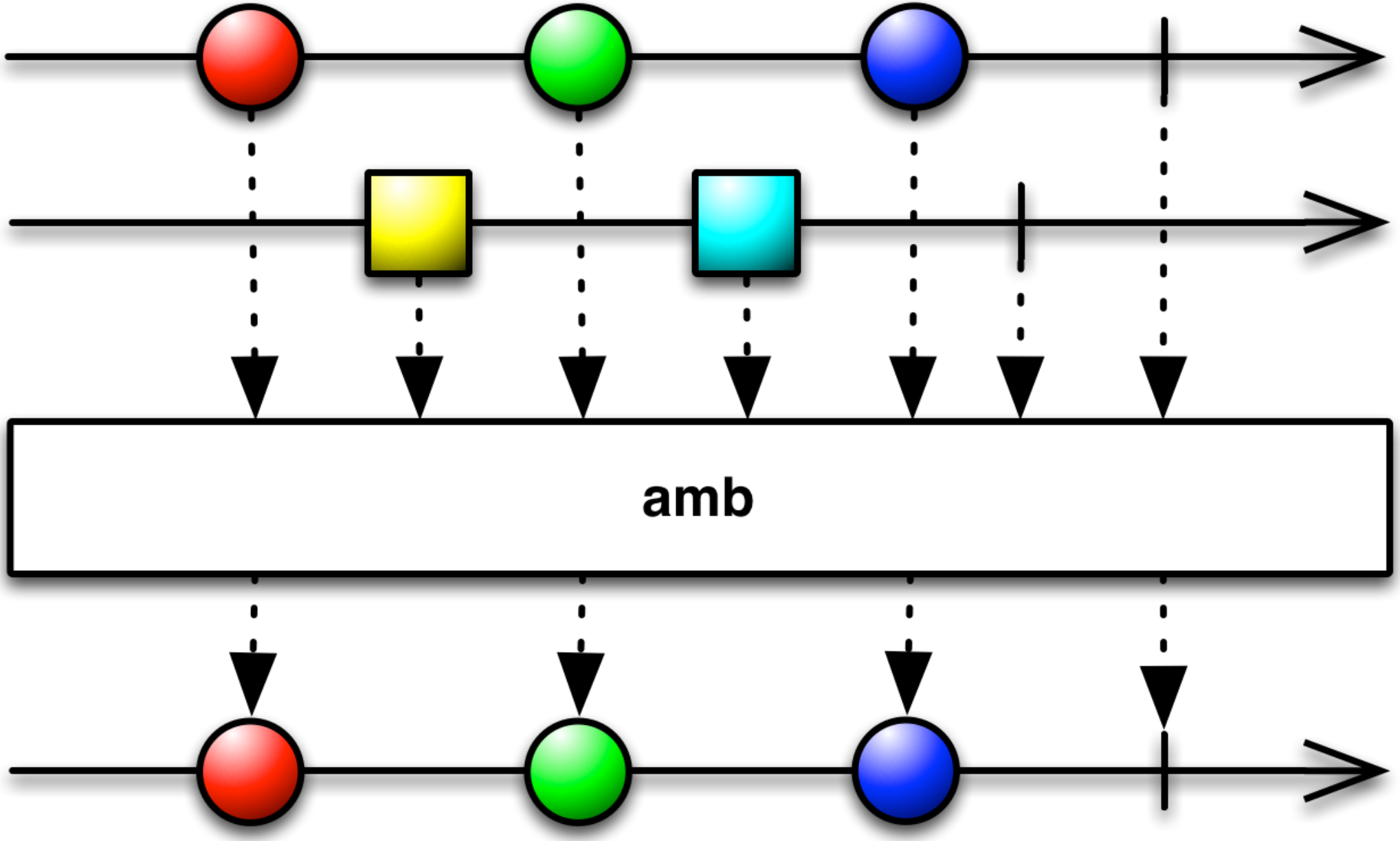
Lab: Standard Functional Operators

Combining Observable Operators


startWith(, ,)

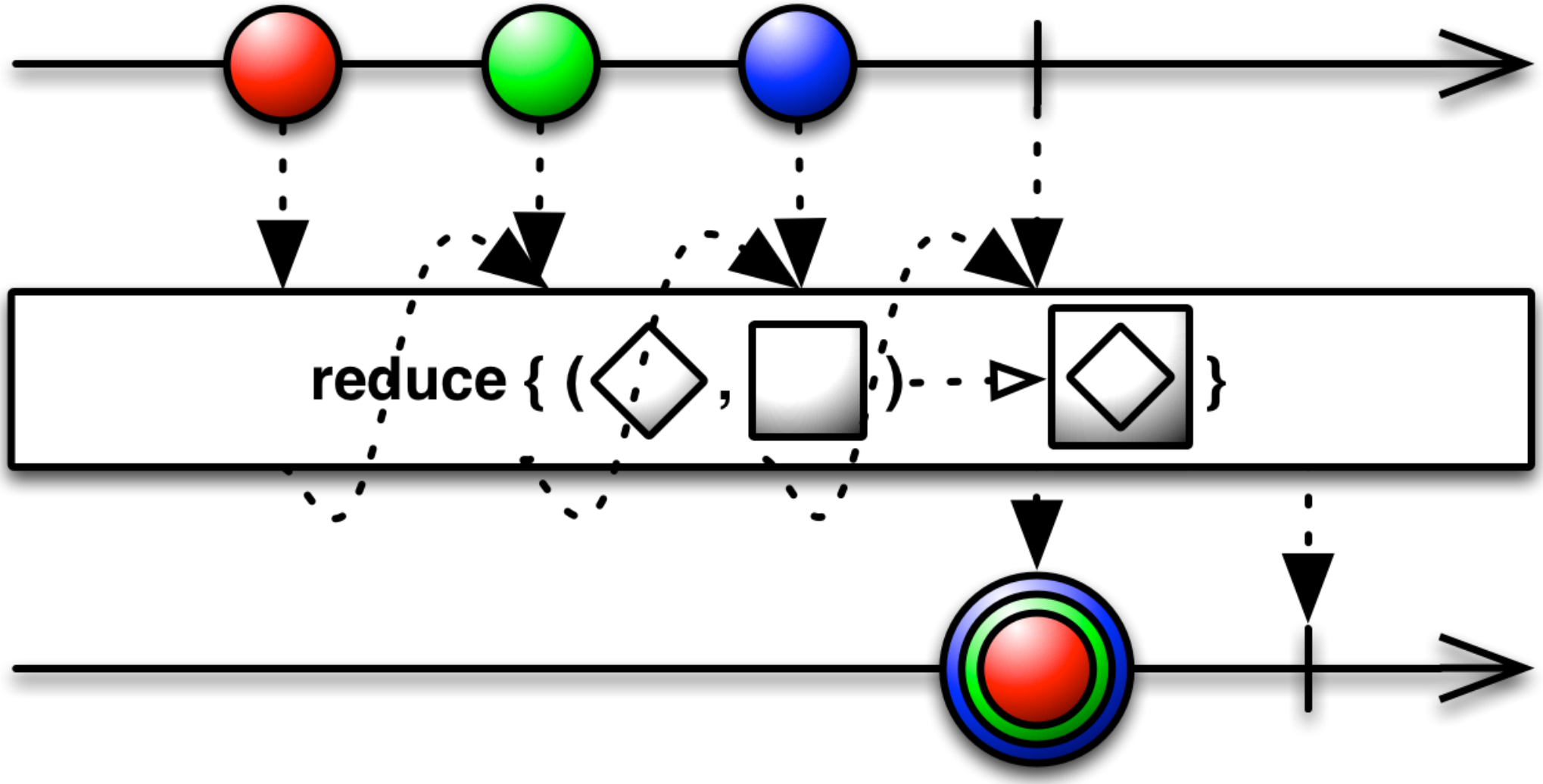

merge

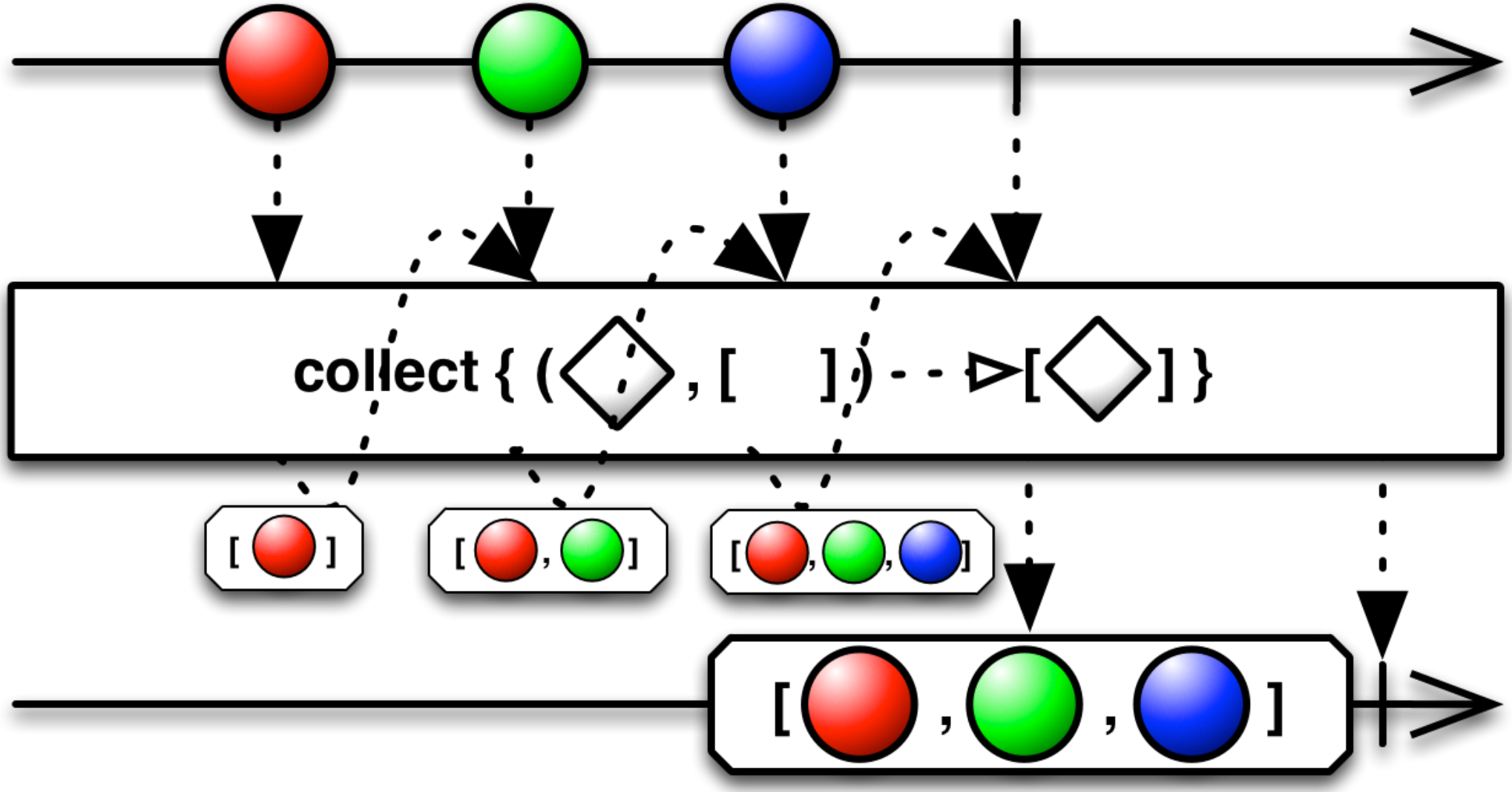

concat

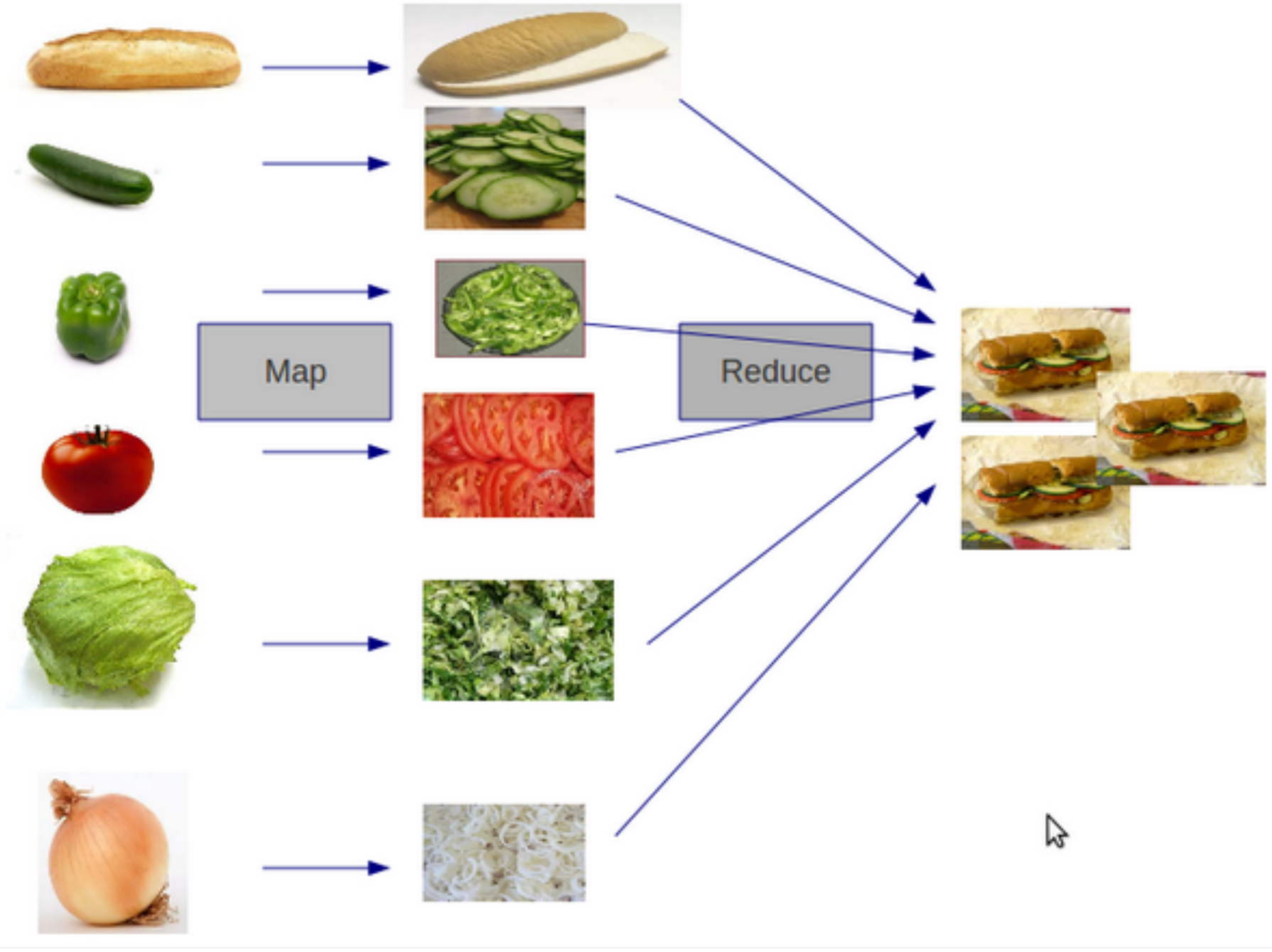

amb

Lab: Combining Observable Operators

Aggregates & Reductions

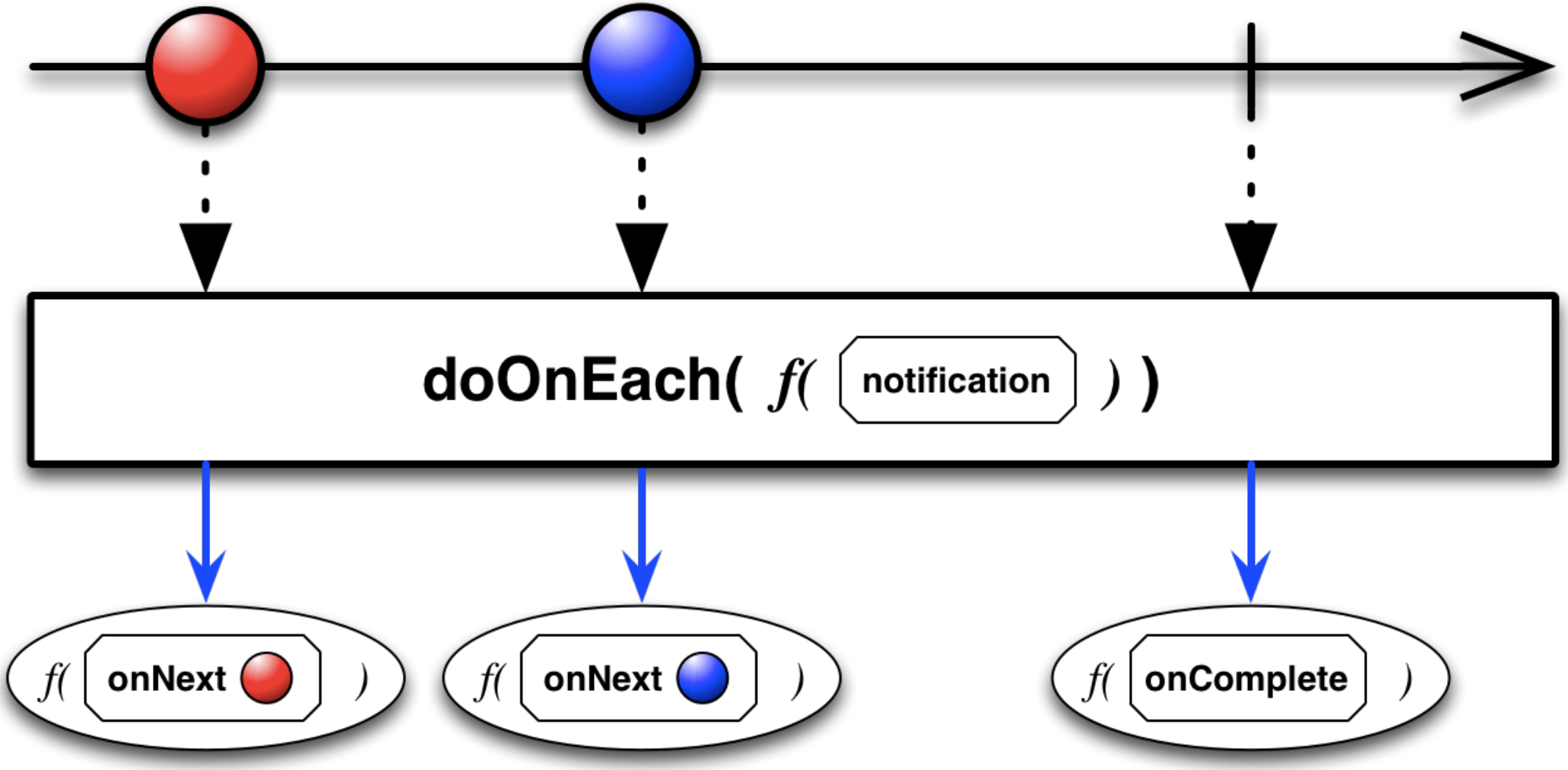

reduce { (,)- - }

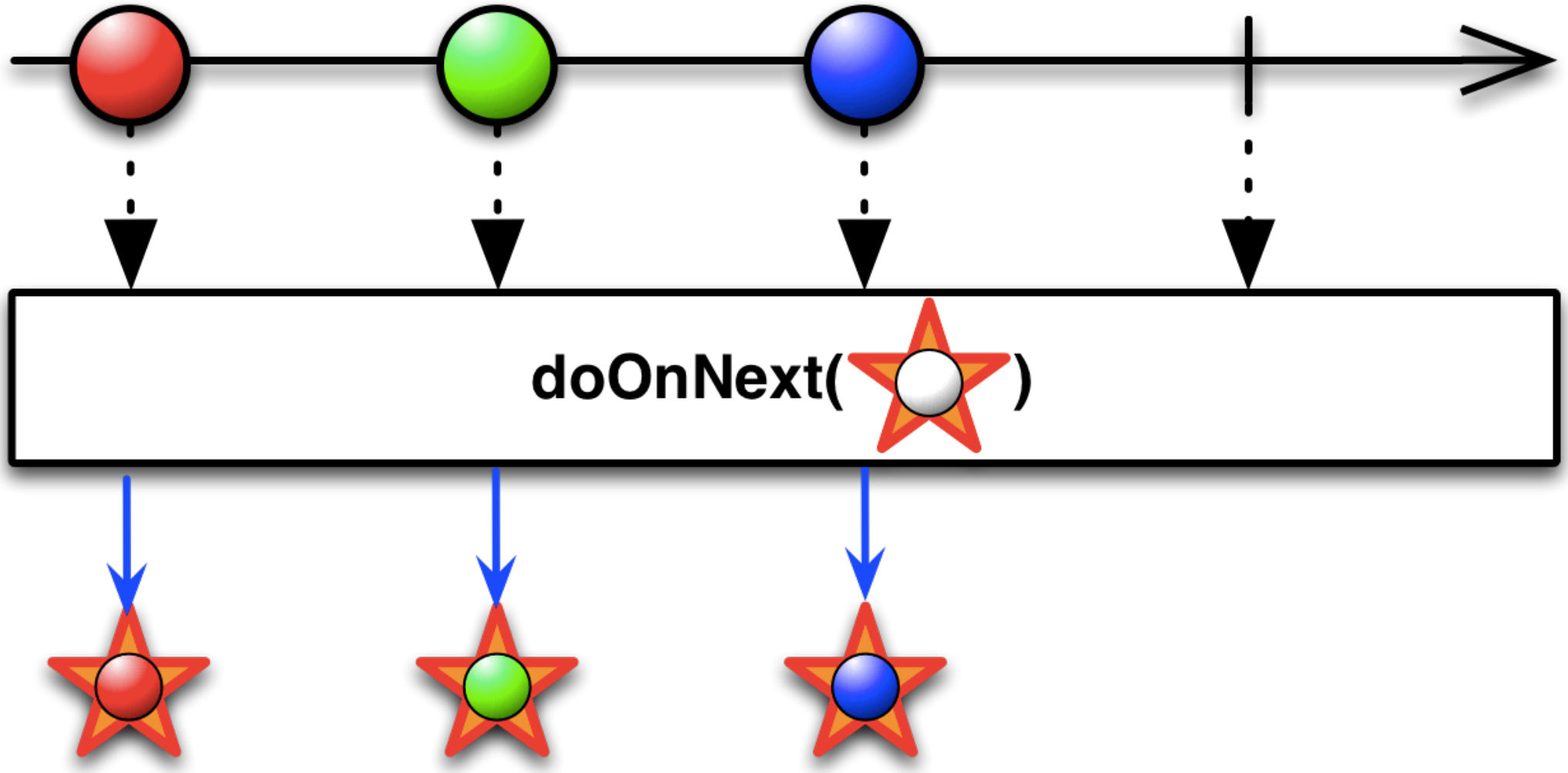

collect { (, [ ]) - - ▷[ ] }
[ ]
[,]
[, ,]
[, ,]

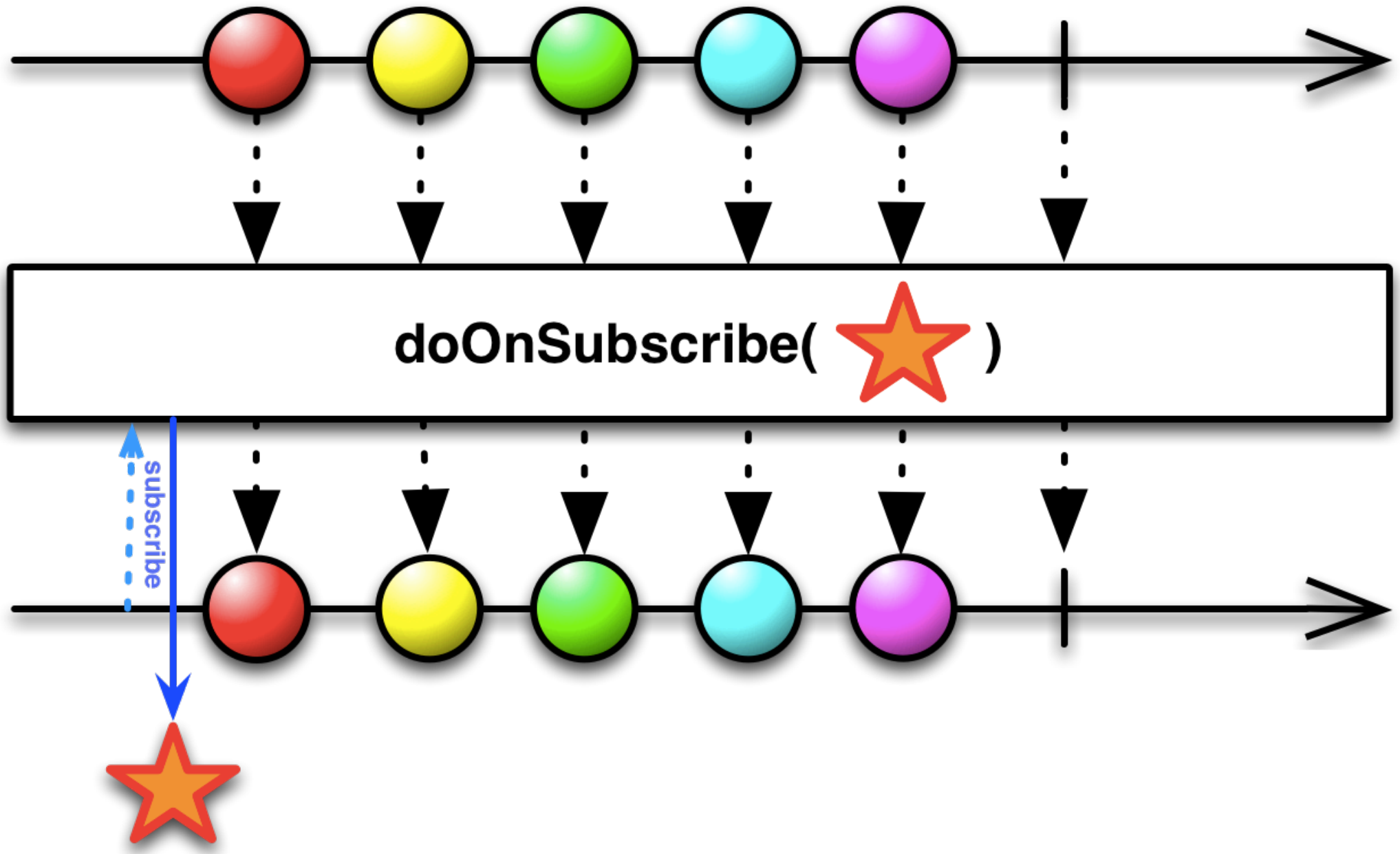

Map
Reduce

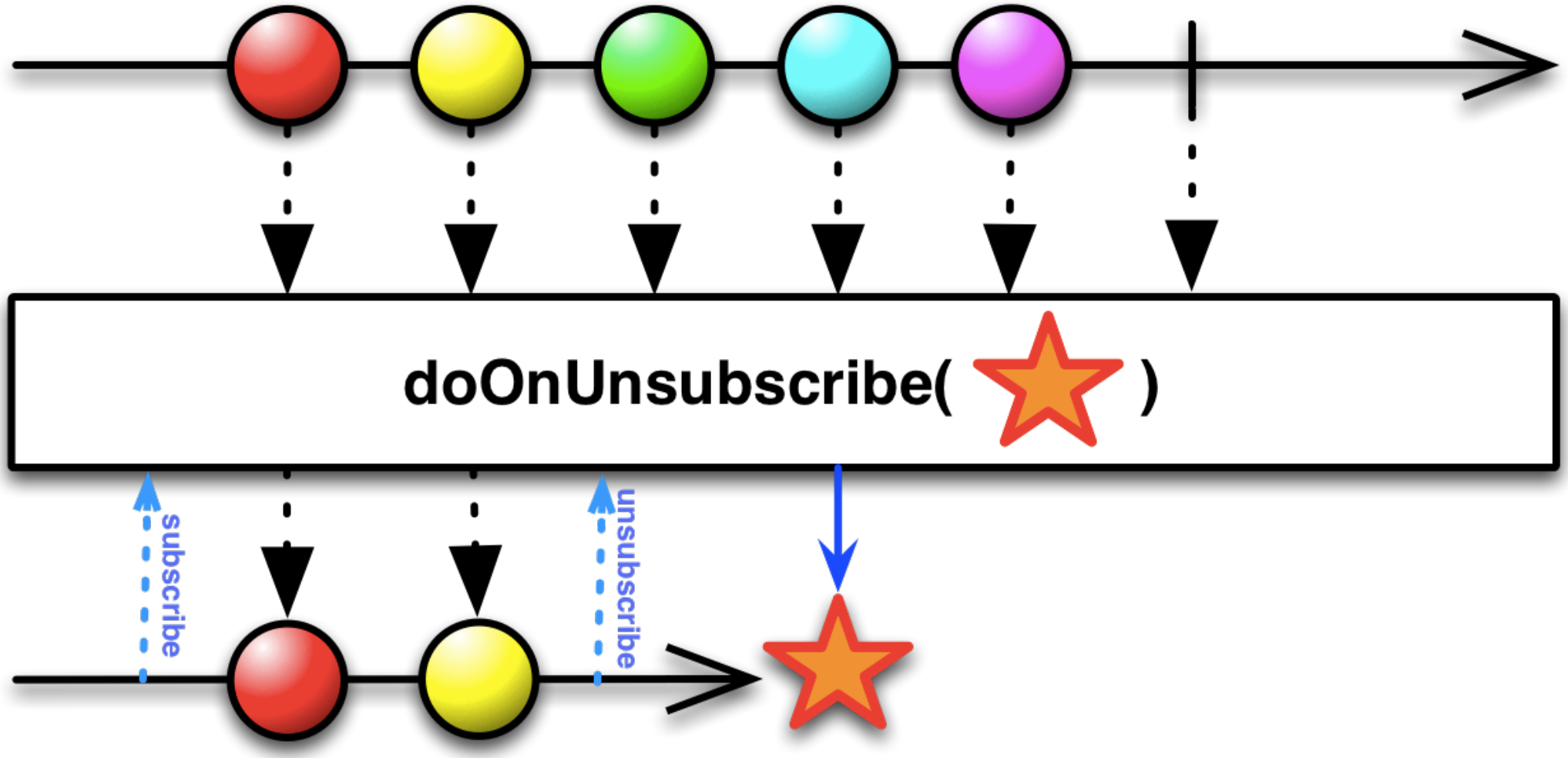
Lab: Aggregates & Reductions

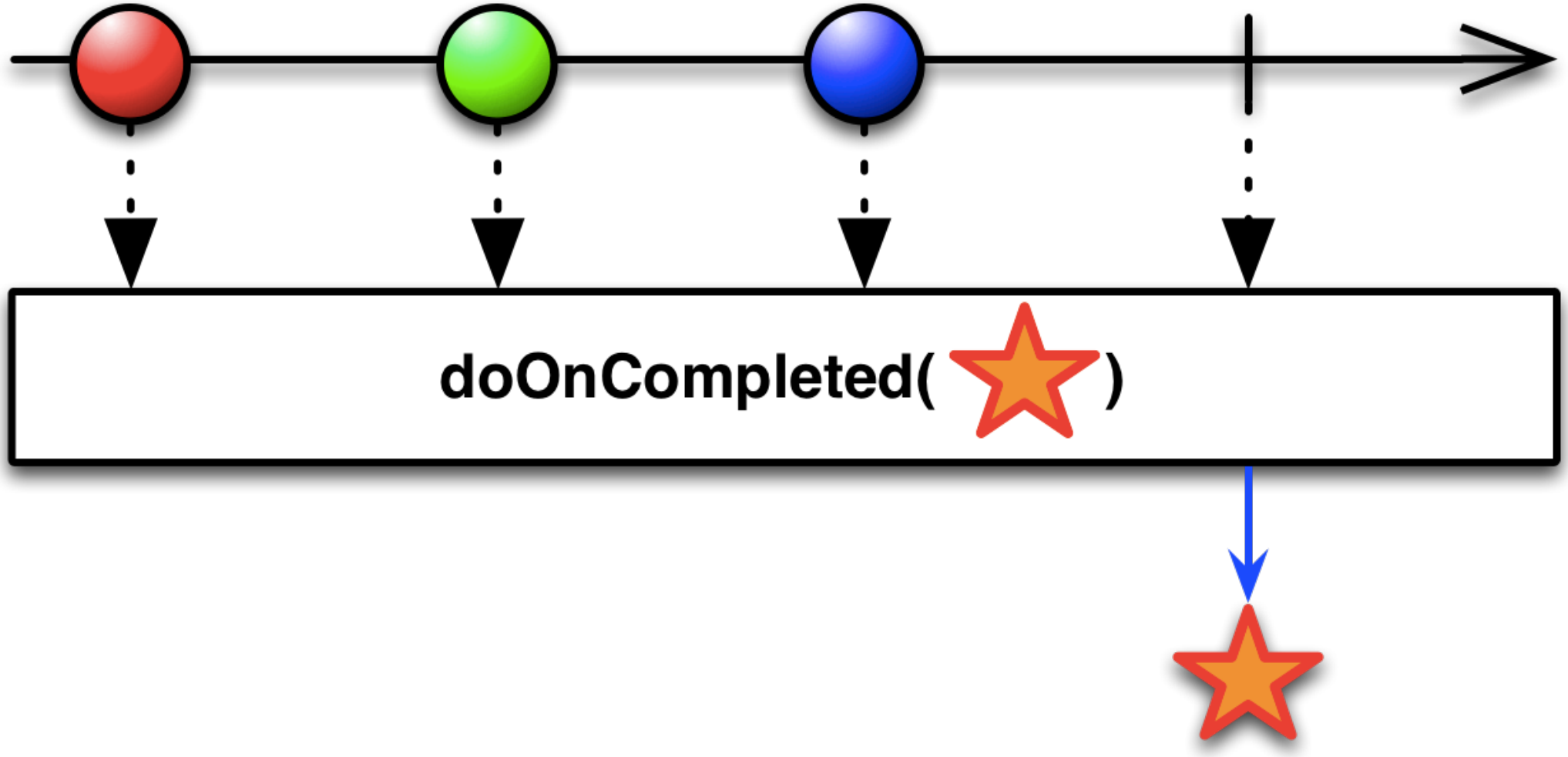
Peeking into the Pipeline

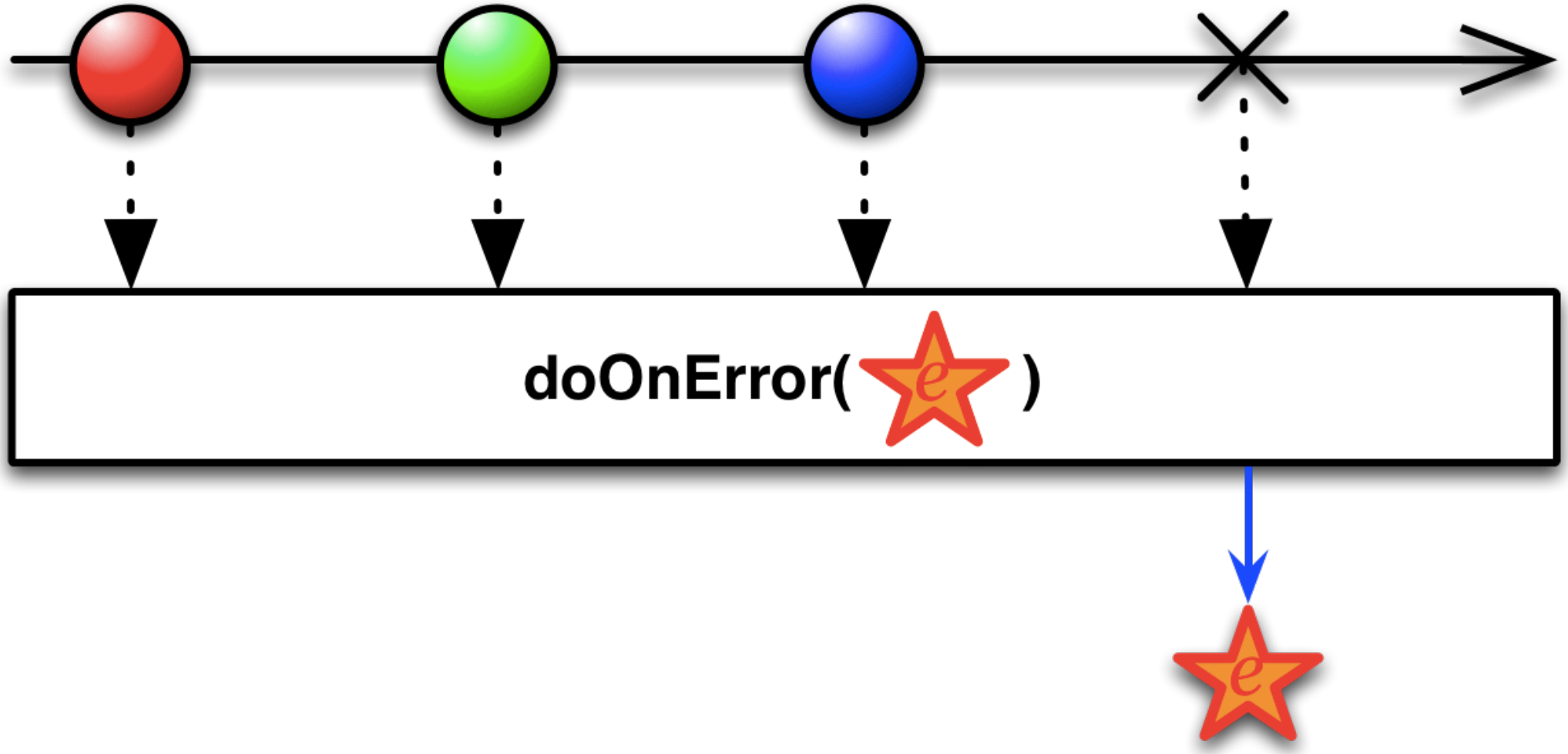

doOnEach(f(notification))
f(onNext)
f(onNext)
f(onComplete)

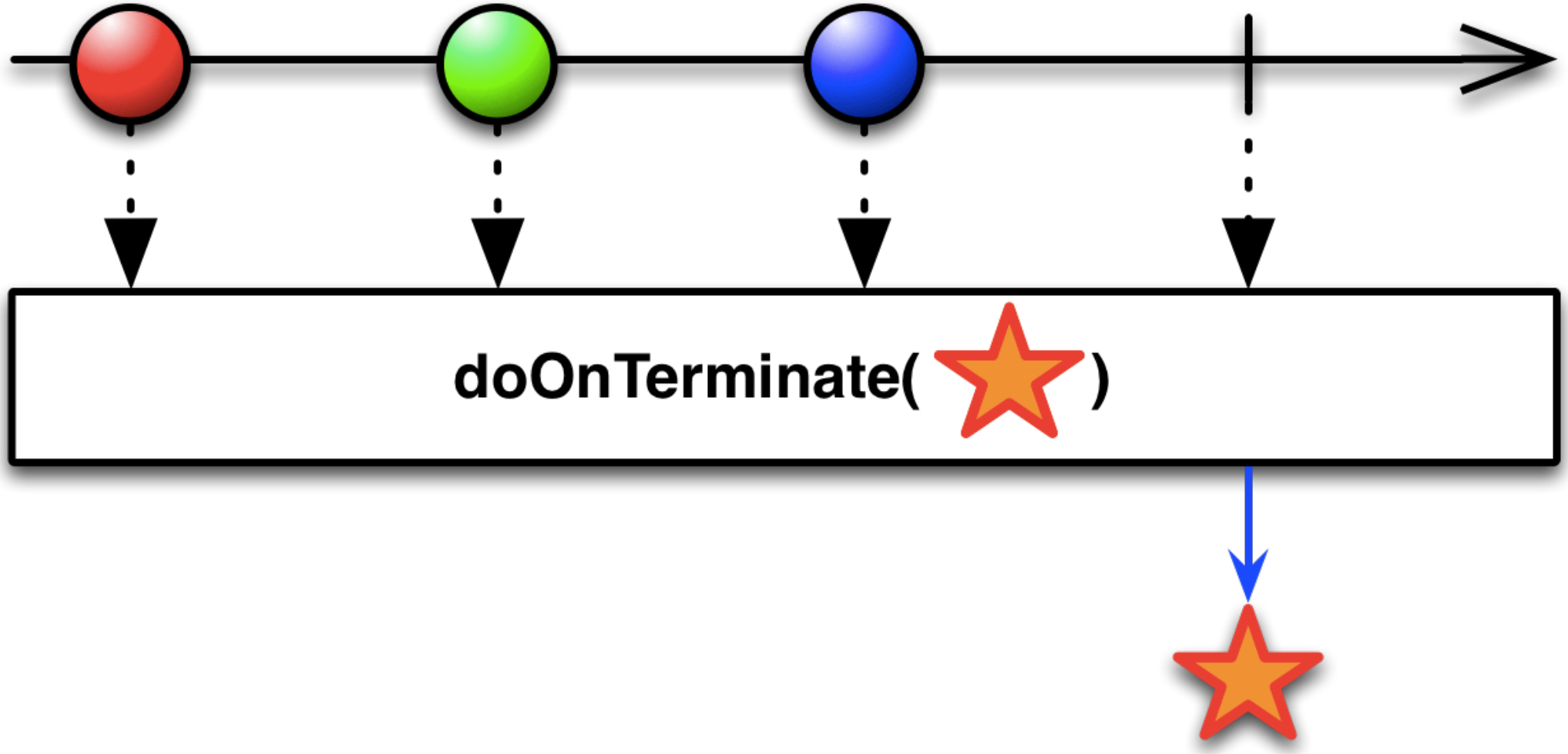

doOnNext( )


doOnSubscribe( )
subscribe

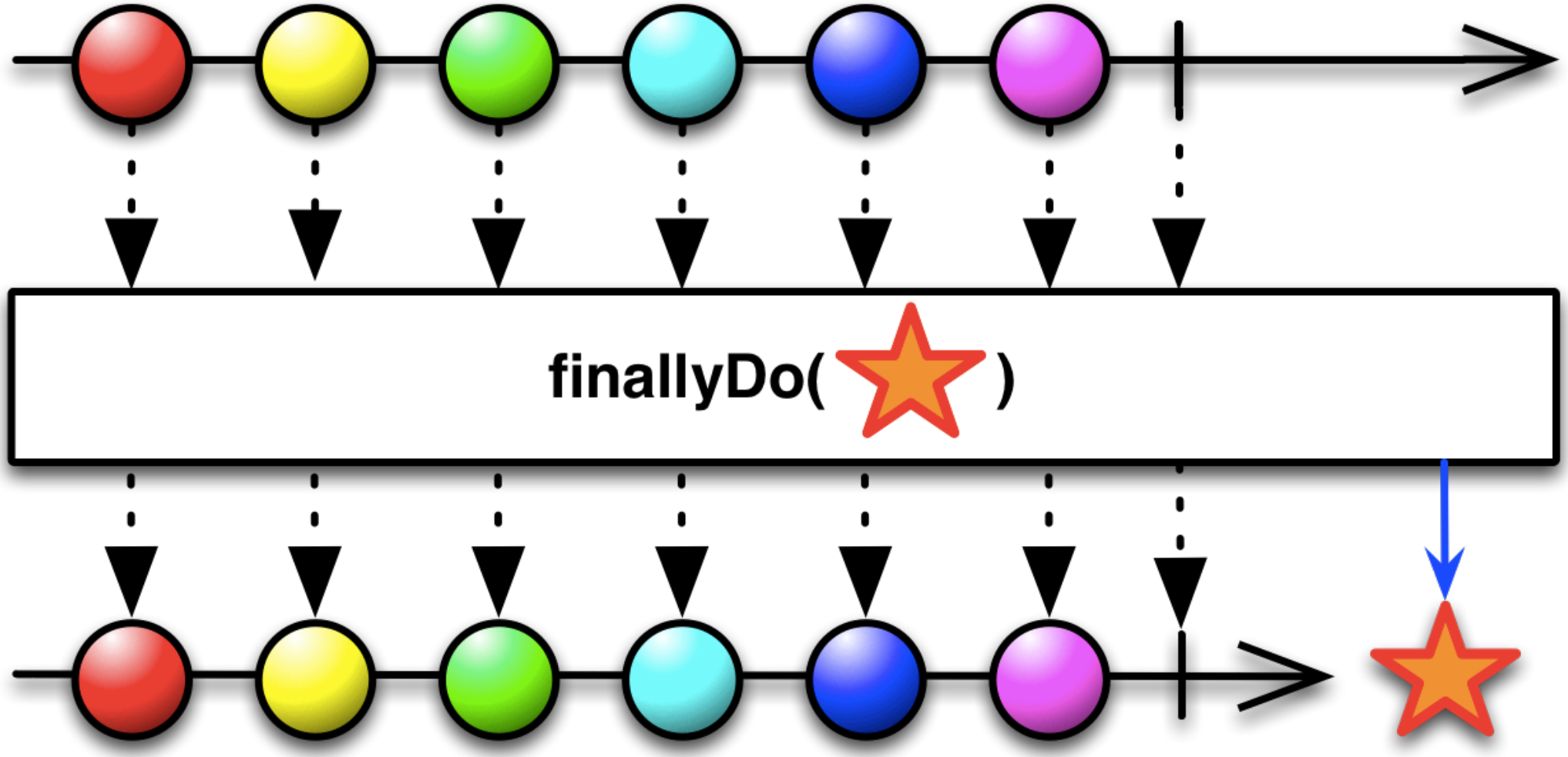

doOnUnsubscribe( )
subscribe
unsubscribe

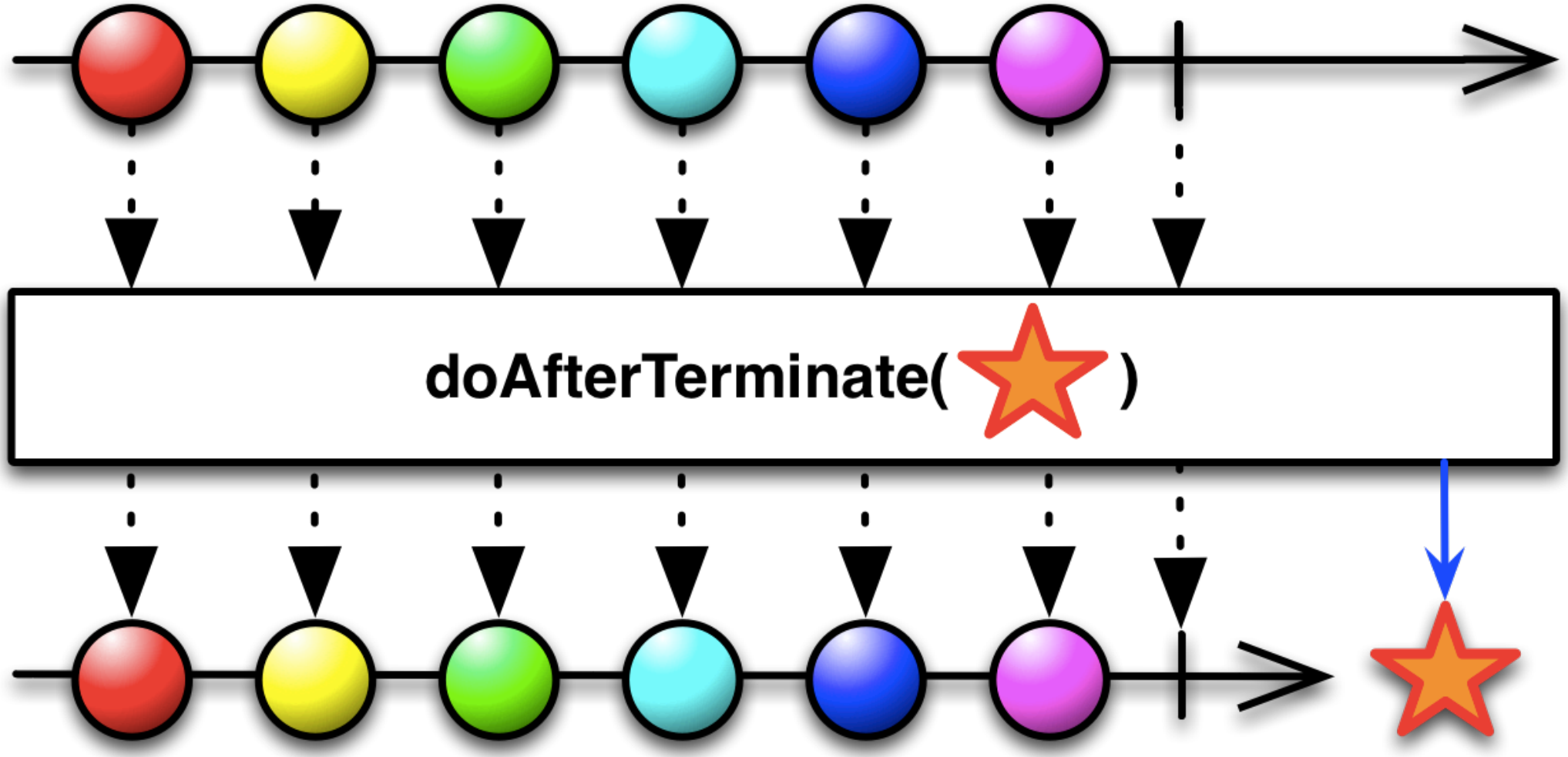

doOnCompleted( )


doOnError(e)
e



Will be called just before the resulting Observable terminates, whether normally or with an error.


finallyDo( )


doAfterTerminate( )

Lab: Peeking into the Pipeline

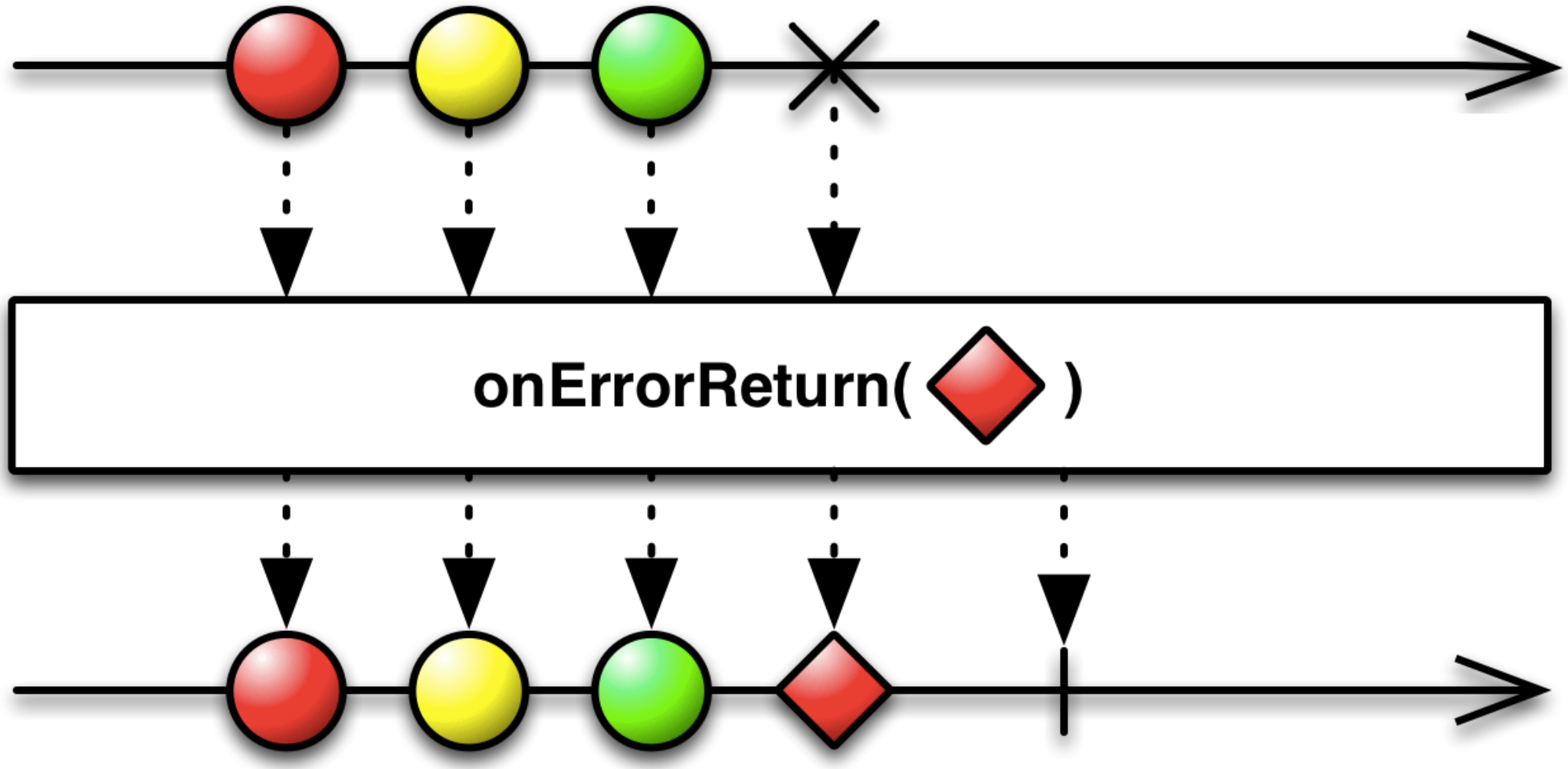
Unit Testing

Testing

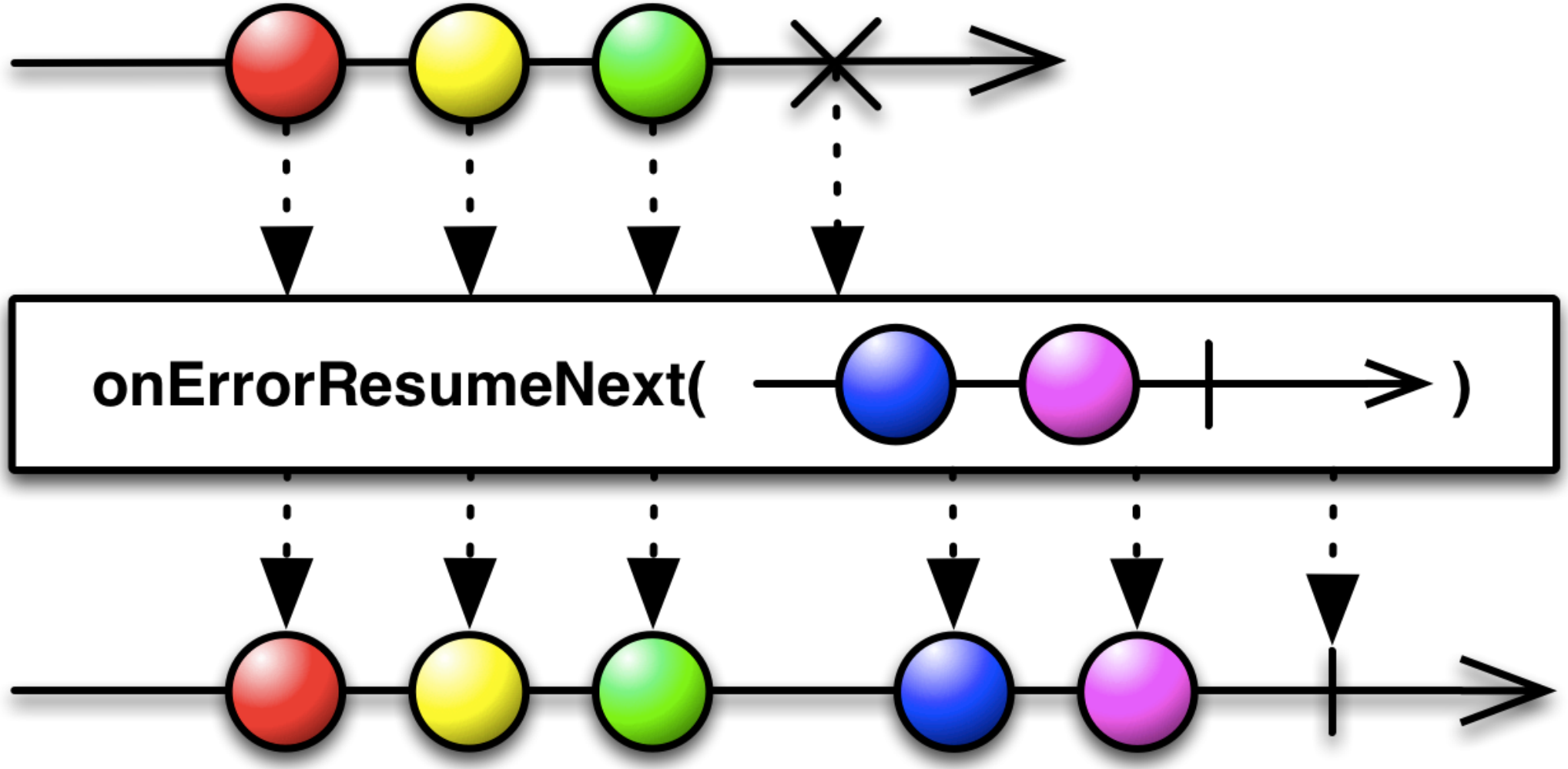
```
Observable.test(); //returns TestObserver

Flowable.test(); //returns testSubscriber

TestSubscriber<Integer> ts = new TestSubscriber<>();

TestObserver<Integer> to = new TestObserver<>();
```

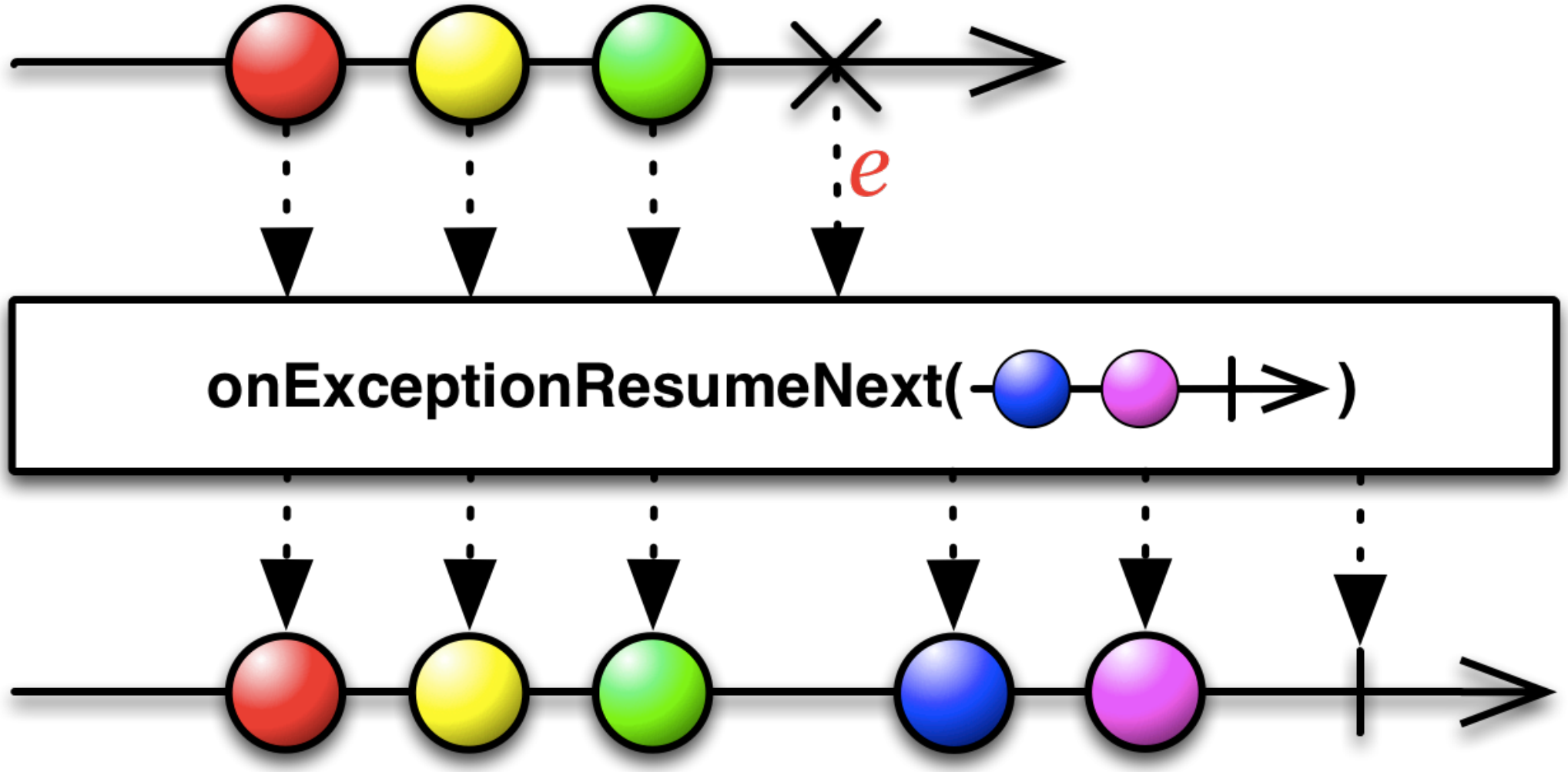
Lab: Unit Testing

Handling Errors



There is now an onErrorReturnItem


onErrorResumeNext(
)


e
onExceptionResumeNext( )

Lab: Handling Errors

Schedulers

RX Single Threads

All threads in RXJava are single threaded by default

RX Scheduler

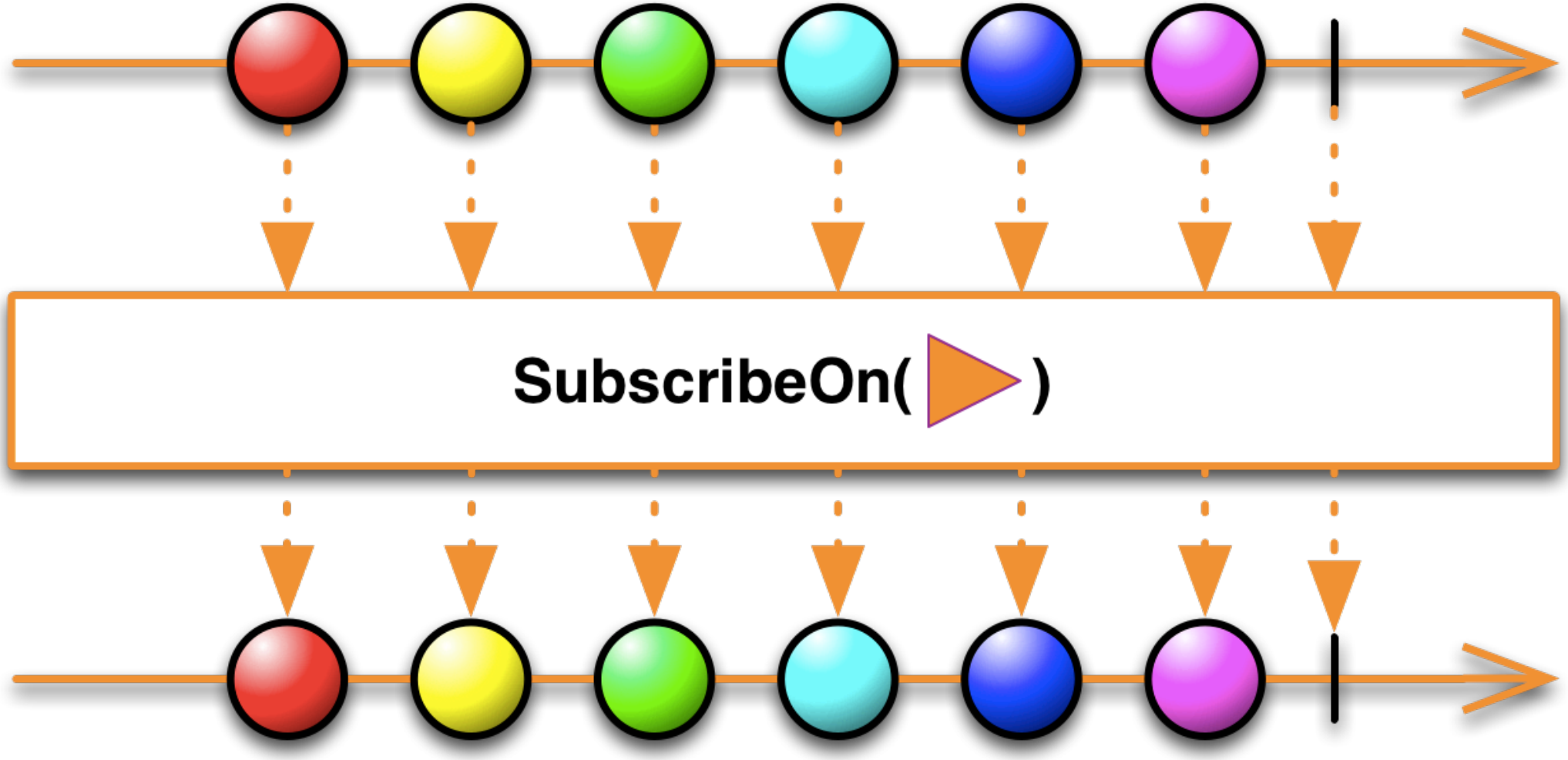
Schedulers assign a different a different Thread for different purposes

RX Scheduler Rule

By default, an `Observable` and the chain of operators that you apply to it will do its work, and will notify its observers, on the same thread on which its `Subscribe` method is called

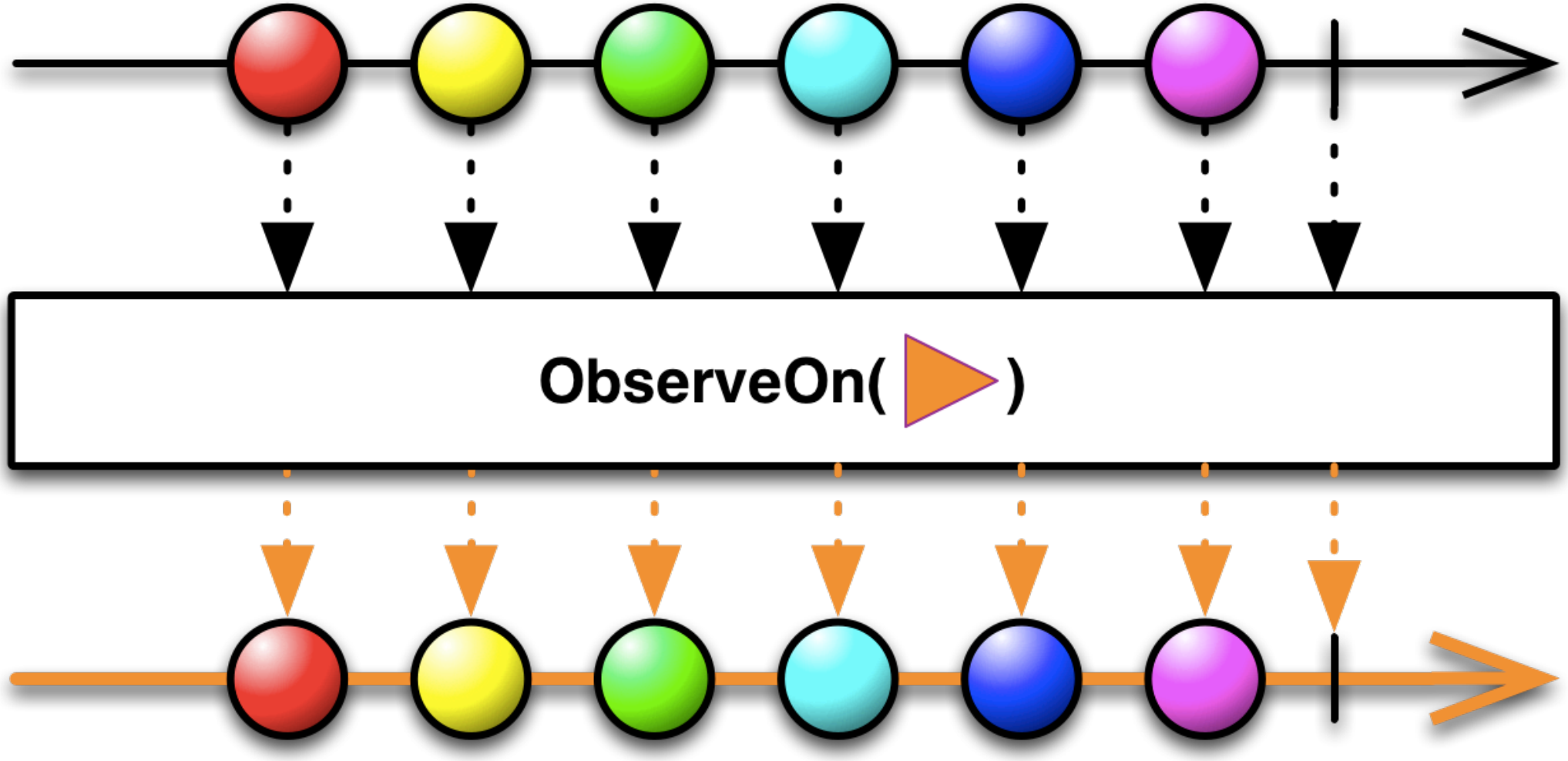
RX subscribeOn()

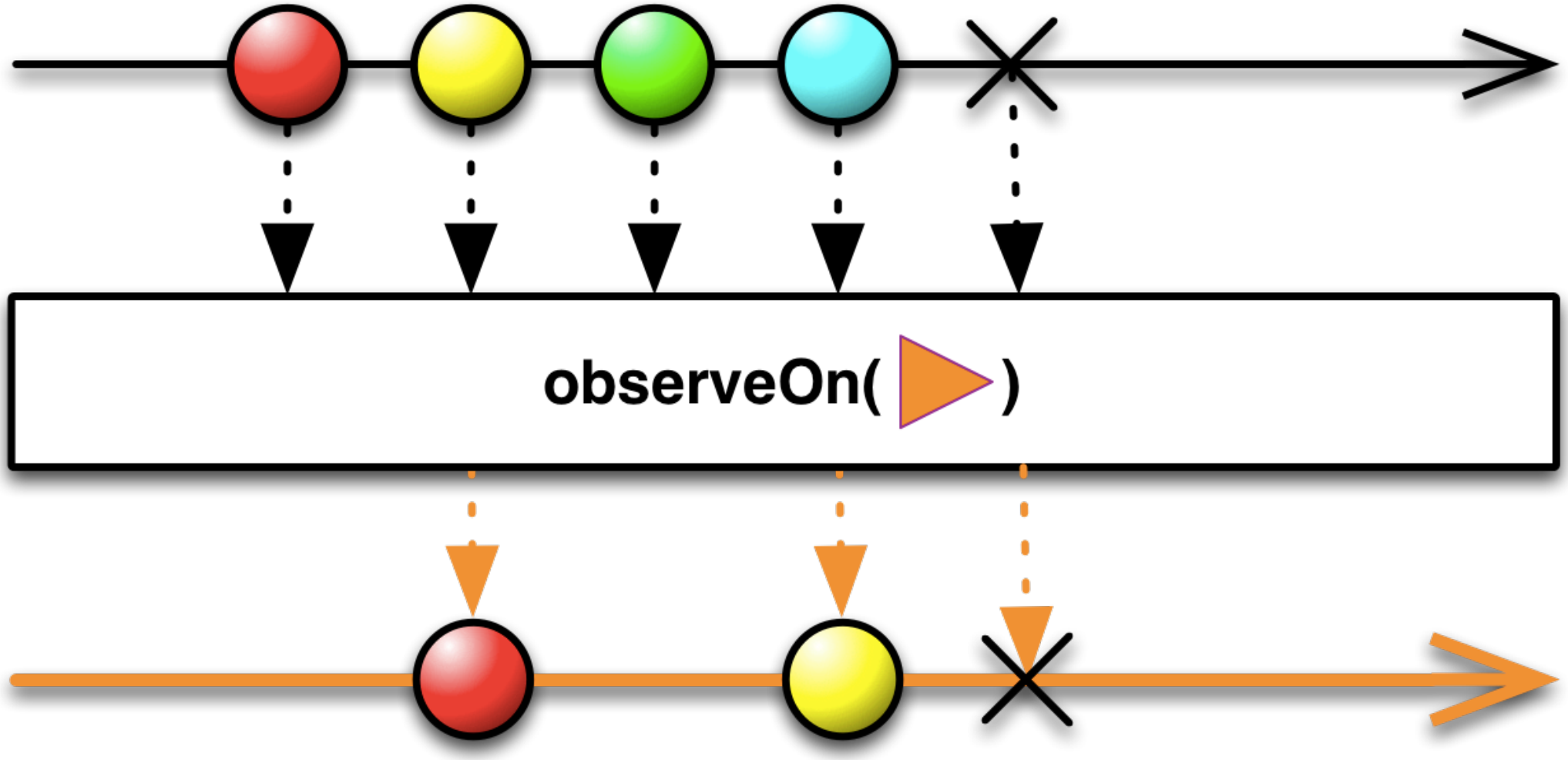
`subscribeOn(...)` to *itself* operate on the specified Scheduler, as well as notifying its observers on that Scheduler.


SubscribeOn( )

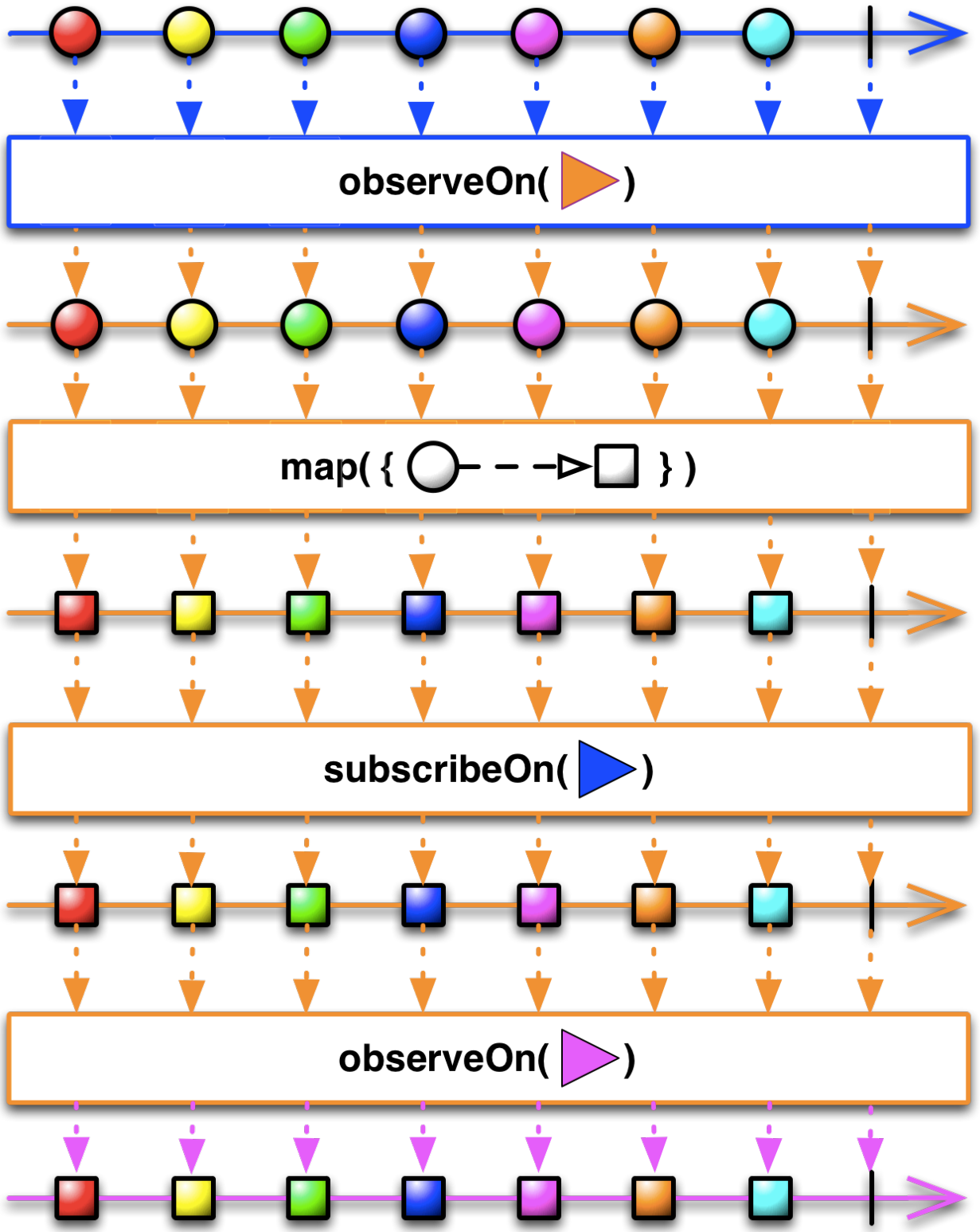
RX observeOn()

`observeOn(...)` assigns a `Scheduler` on the observable side of invocation


ObserveOn( )



If an error is received, that error will be sent to the asynchronous observable/flowable immediately


observeOn( )
map({ })
subscribeOn( )
observeOn( )

Lab: ObserveOn & SubscribeOn

RX immediate()

Creates and returns a Scheduler that executes work immediately on the current thread *synchronously*.

RX trampoline()

Creates and returns a Scheduler that queues work on the current thread to be executed after the current work completes. One by one, perfect for funneling jobs to a process that can only take requests one at a time.

RX newThread()

Creates and returns a Scheduler that creates a new Thread for each unit of work.

RX computation()

Creates and returns a `Scheduler` intended for computational work. This can be used for event-loops, processing callbacks and other computational work. Use for *Non-blocking* tasks

RX computation()

Do not perform IO-bound work on this scheduler. Use `Schedulers.io()` instead.

RX computation()

Backed by a bounded `ScheduledExecutorService` thread-pool with size equal to the number of available processors.

RX computation()

Avoid big performance hits by thread creation overhead and context switching overhead

http://stackoverflow.com/questions/31276164/rxjava-schedulers-use-cases

RX io()

Creates and returns a Scheduler intended for IO-bound work. This can be used for asynchronously performing blocking IO. Do not perform computational work on this scheduler. Use `Schedulers.computation()` instead.

RX io()

The implementation is backed by an Executor thread-pool that will grow as needed.

RX io()

This can be used for asynchronously performing blocking IO.

RX io()

Do not perform computational work on this scheduler. Use `Schedulers.computation()` instead.

RX Android Schedulers

Android based Schedulers.
`mainThread(...)` to perform tasks on the Android main thread.

https://github.com/ReactiveX/RxAndroid

RX JavaFX Schedulers

`JavaFxScheduler.getInstance()` used to update UI components on the JavaFX Event Dispatch Thread

https://github.com/ReactiveX/RxJavaFX

Lab: Schedulers


UNDER BACKPRESSURE

Backpressure *def.*

Back pressure refers to pressure opposed to the desired flow of a fluid in a confined place such as a pipe. It is often caused by obstructions or tight bends in the confinement vessel along which it is moving, such as piping or air vents.

Backpressure *def.*

Back pressure is when an `Flowable` is emitting items more rapidly than an operator or subscriber can consume them. This presents the problem of what to do with such a growing backlog of unconsumed items.

Lab: Backpressure

Hot & Cold Observables

PSY- Gangnam Style (Official Music Video)

DanceGangnamStyle

Subscribe 46,824

44,983,176

Add to Share More

88,792 23,430

Southwest Florida Eagle Cam

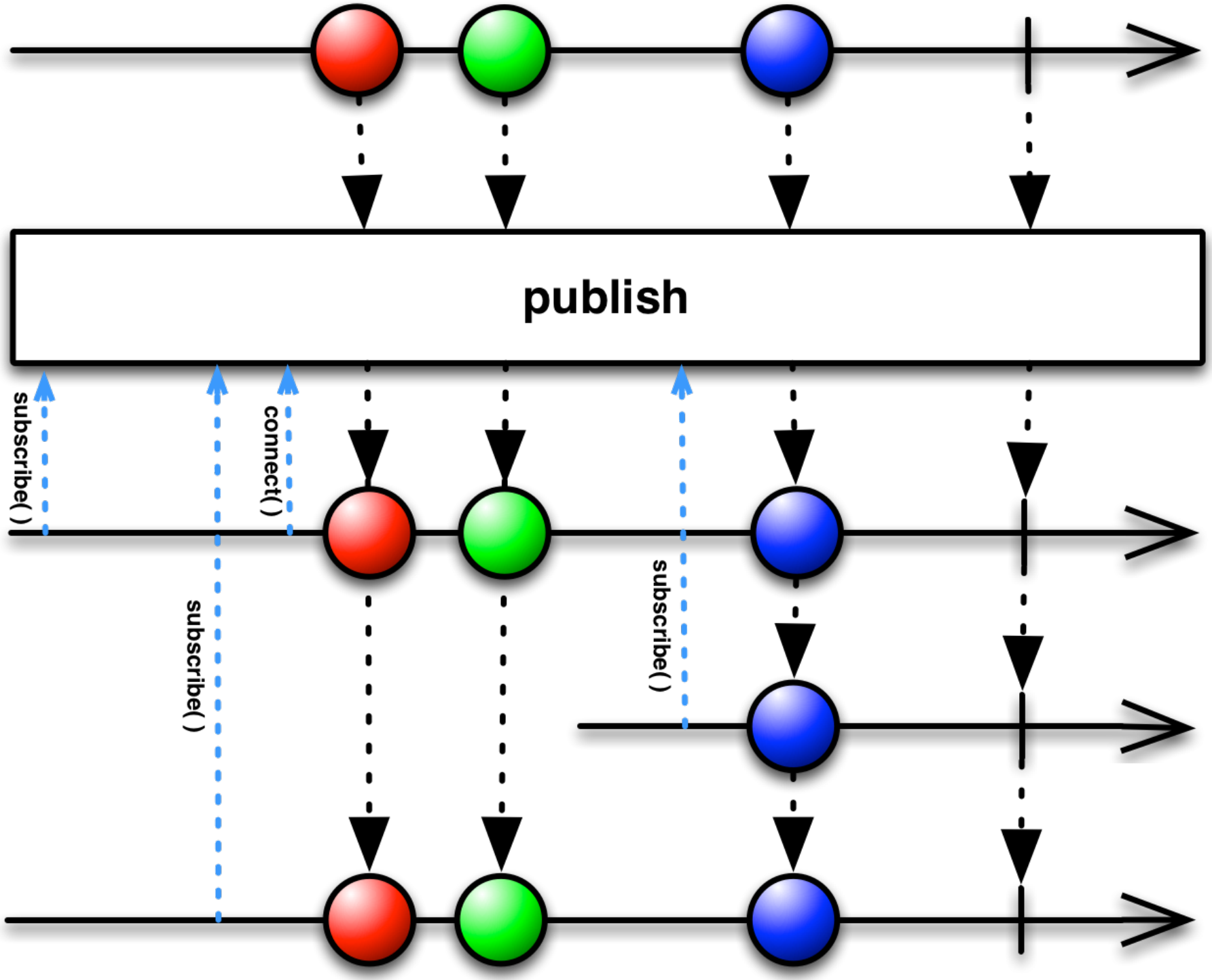
Southwest Florida Eagle Cam

Subscribe 6,394

2592 watching now

Add to Share More

4,599 392


publish
subscribe( )
connect( )
subscribe( )
subscribe( )

RX ConnectableObservables

`publish()` returns a `ConnectableObservable`

RX autoConnect

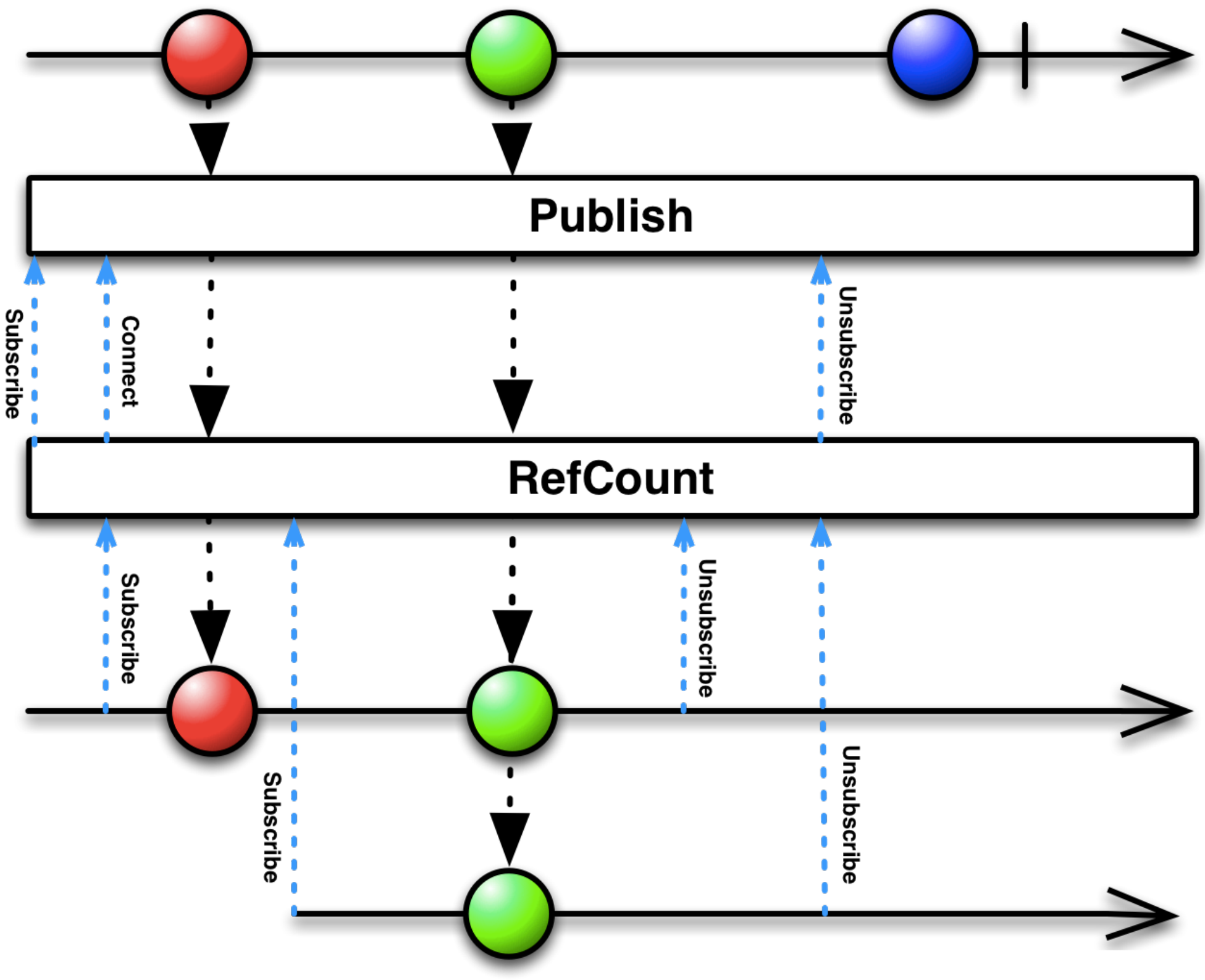
`autoConnect()`
Returns an Observable that automatically connects to this ConnectableObservable when the first Subscriber subscribes.

RX connect

`connect()`
Instructs the `ConnectableObservable` to begin emitting the items from its underlying `Observable` to its `Subscribers`.

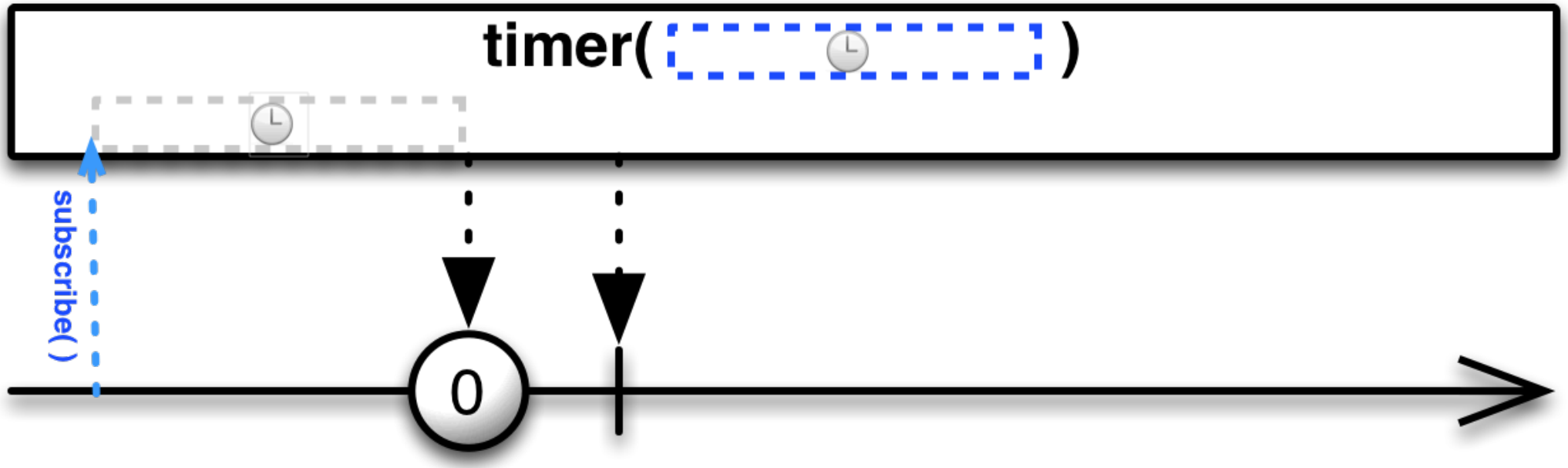
RX refCount()

`refCount()`
Returns an `Observable` that stays connected to this `ConnectableObservable` as long as there is at least one subscription to this `ConnectableObservable.`

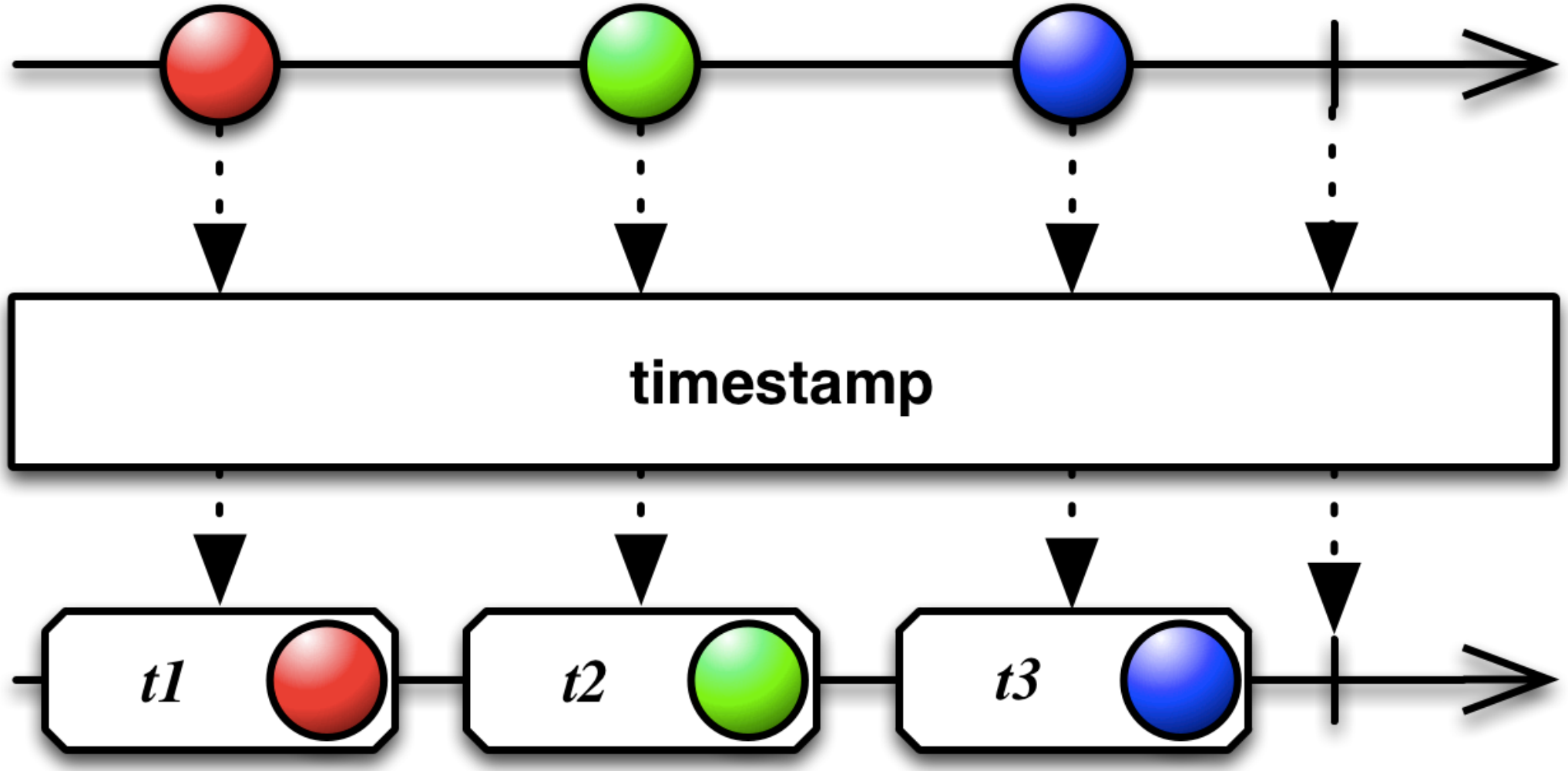

Publish
Subscribe
Connect
Unsubscribe
RefCount
Subscribe
Unsubscribe
Subscribe
Unsubscribe

Lab: Hot/Cold Observables

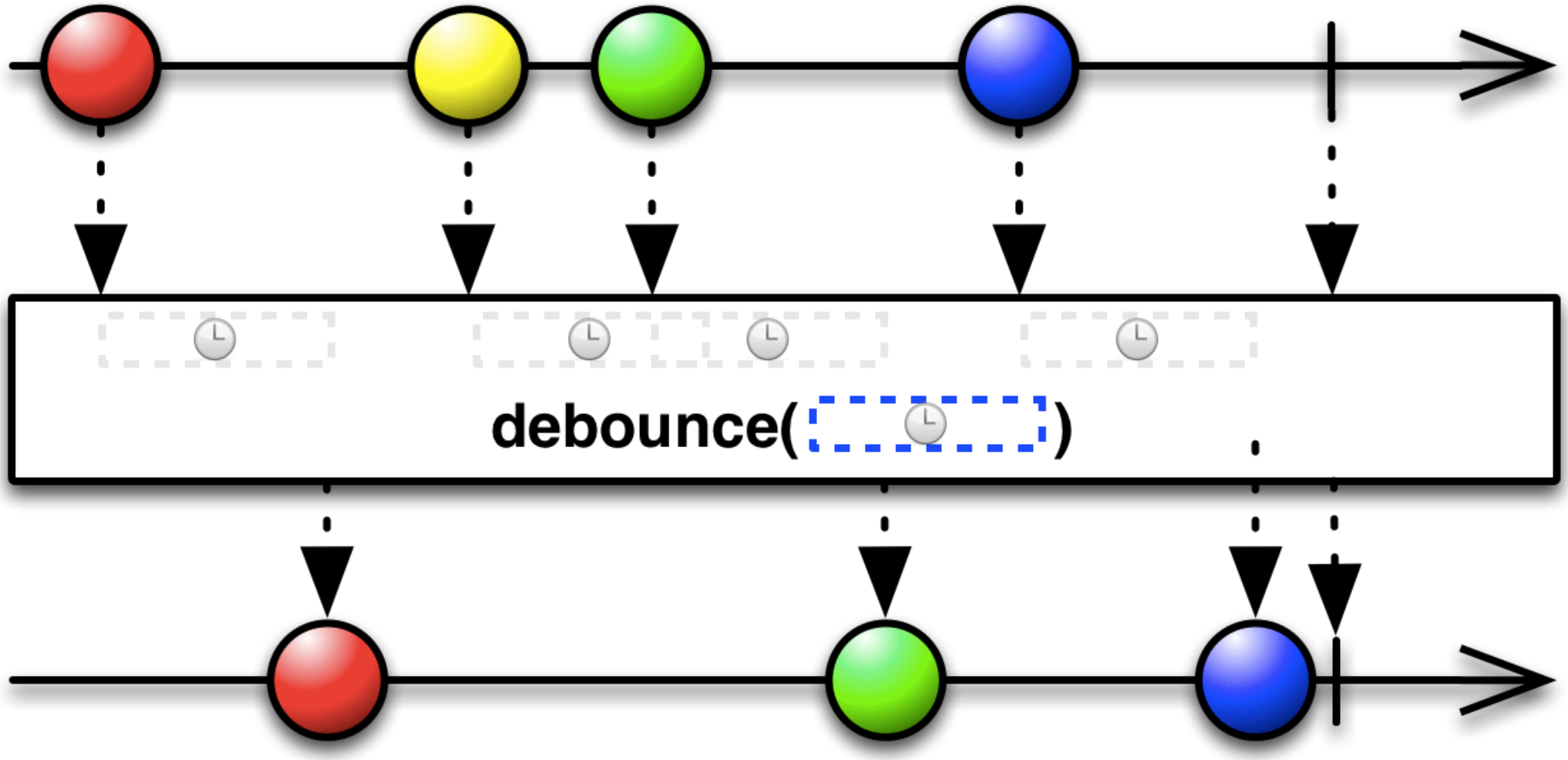
Time Remaining Labs: Interesting Operators



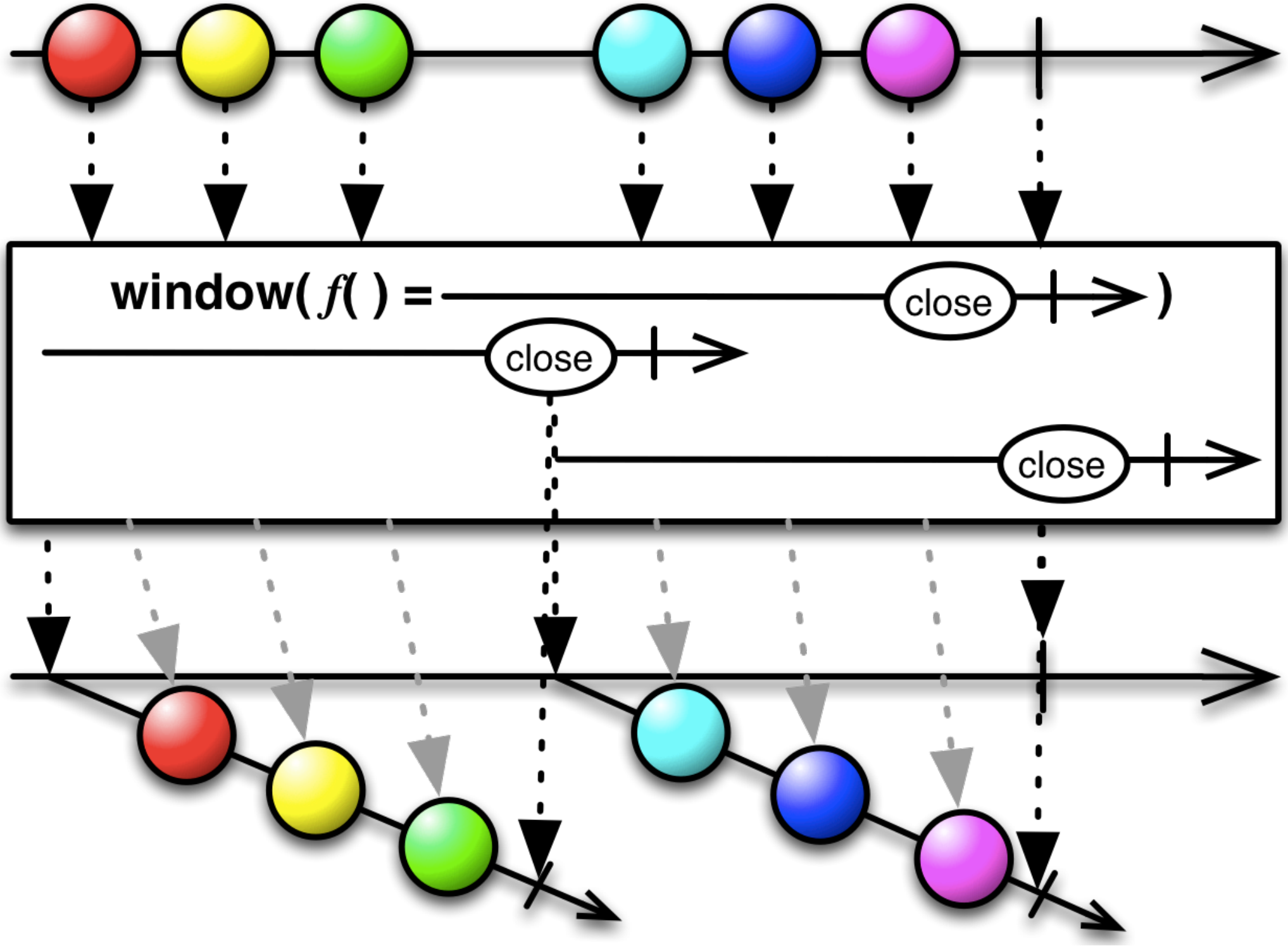
Operates on the computation Scheduler

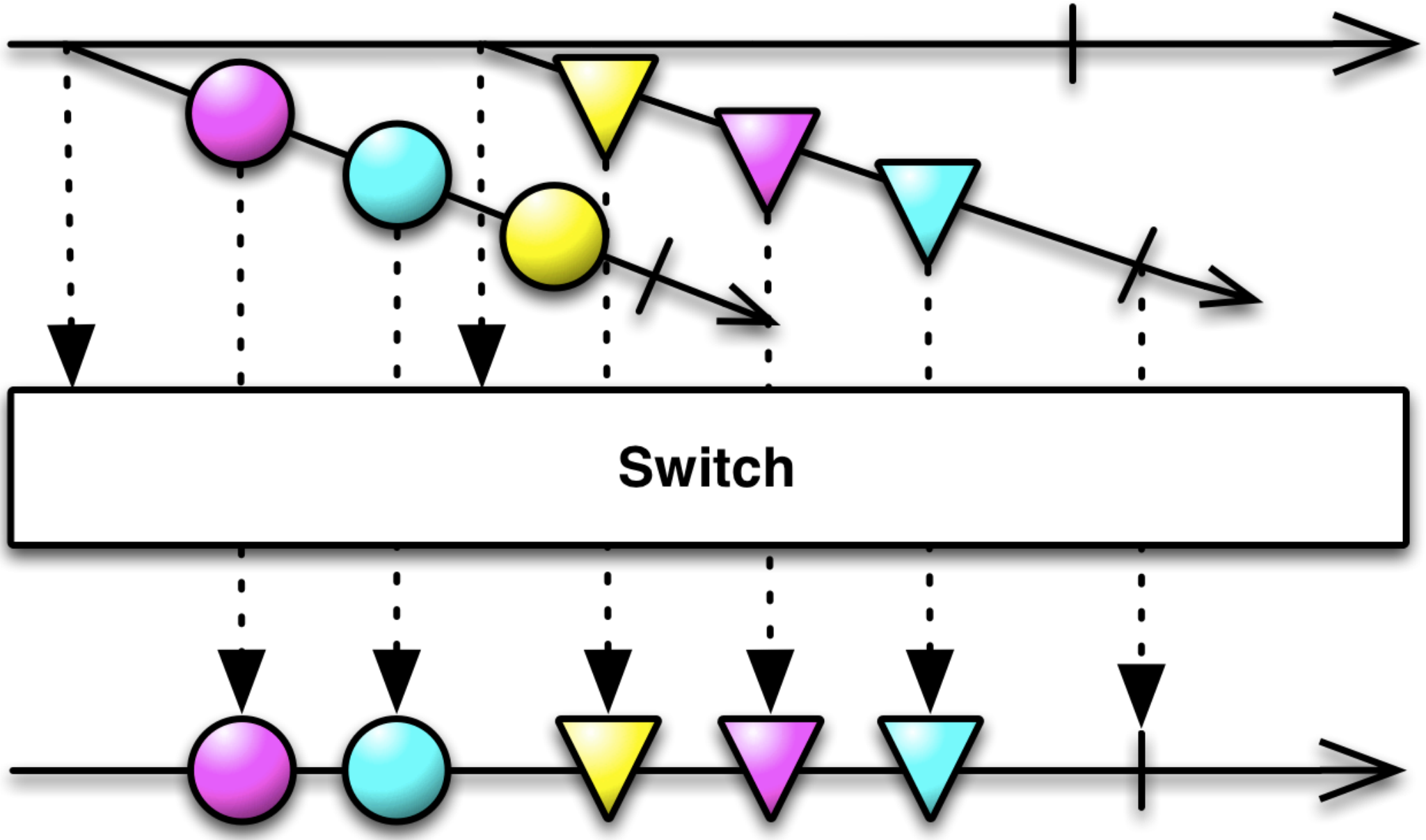


Operates on the immediate Scheduler



Operates on the computation Scheduler


window(f( ) =
close
)
close
close


Switch

Thank You