ReactiveCocoa

INTRODUCTION

ILYA LARYIONAU

@LARRYONOFF

COMPONENTS

- EVENTS
- OBSERVERS
- DISPOSABLES
- SIGNALS
- SIGNAL PRODUCERS
- SCHEDULERS

EVENTS

REPRESENTATION OF THE FACT THAT SOMETHING HAS HAPPENED

EVENTS

```
enum Event<Value, Error: ErrorType> {
    case Next(Value)
    case Failed(Error)
    case Completed
    case Interrupted
}
```

OBSERVERS

AN OBSERVER IS ANYTHING THAT IS WAITING OR CAPABLE OF WAITING FOR EVENTS

OBSERVERS

```
struct Observer<Value, Error: ErrorType> {
    func sendNext(value: Value)
    func sendFailed(error: Error)
    func sendCompleted()
    func sendInterrupted()
}
```

DISPOSABLES

A DISPOSABLE IS A MECHANISM FOR MEMORY MANAGEMENT AND CANCELLATION.

DISPOSABLES REPRESENTED BY

- SIMPLEDISPOSABLE
- ACTIONDISPOSABLE
- COMPOSITEDISPOSABLE
- SCOPEDDISPOSABLE
- SERIALDISPOSABLE

SIMPLEDISPOSABLE

A DISPOSABLE THAT ONLY FLIPS DISPOSED UPON DISPOSAL. AND PERFORMS NO OTHER WORK

ACTIONDISPOSABLE

A DISPOSABLE THAT WILL RUN AN ACTION UPON DISPOSAL

COMPOSITEDISPOSABLE

A DISPOSABLE THAT WILL DISPOSE OF ANY NUMBER OF OTHER DISPOSABLES

SCOPEDDISPOSABLE

A DISPOSABLE THAT, UPON DEINITIALIZATION, WILL AUTOMATICALLY DISPOSE OF ANOTHER DISPOSABLE

SERIALDISPOSABLE

A DISPOSABLE THAT WILL OPTIONALLY DISPOSE OF ANOTHER DISPOSABLE

`innerDisposable`

SIGNALS

A SIGNAL IS SERIES OF EVENTS OVER TIME THAT CAN BE OBSERVED

SIGNALS

OBSERVE*

```
final class Signal<Value, Error: ErrorType> {
  func observeNext(next: Value -> ()) -> Disposable?
  func observeCompleted(completed: () -> ()) -> Disposable?
  func observeFailed(error: Error -> ()) -> Disposable?
  func observeInterrupted(interrupted: () -> ()) -> Disposable?
}
```

PIPES

A SIGNAL THAT CAN BE MANUALLY CONTROLLED

PIPES

```
final class Signal<Value, Error: ErrorType> {
  static func pipe() -> (Signal, Observer)
}
```

PIPES

```
let (signal, observer) = Signal<Int, NoError>.pipe()

signal.observeNext { intValue in print("\(intValue)") }

observer.sendNext(1) // 1
observer.sendNext(2) // 2
observer.sendNext(3) // 3

observer.sendCompleted()
```

SIGNAL PRODUCERS

A SIGNAL PRODUCER IS ANY SERIES OF EVENTS OVER TIME THAT CAN BE OBSERVED. BUT HAS TO BE STARTED

SIGNAL PRODUCERS

```
struct SignalProducer<Value, Error: ErrorType> {
  init<S: SignalType where S.Value == Value, S.Error == Error>(signal: S)
  init(_ startHandler: (Signal<Value, Error>.Observer, CompositeDisposable) -> ())
  init(value: Value)
  init(error: Error)
  init<S: SequenceType where S.Generator.Element == Value>(values: S)
}
```

SIGNAL PRODUCERS

```
let oneProducer = SignalProducer<Int, NoError>(value: 1)

// .Next(1)
// .Completed
```

SIGNAL PRODUCERS

```
let oneTwoThreeProducer = SignalProducer<Int, NoError>(values: [1, 2, 3])

// .Next(1)
// .Next(2)
// .Next(3)
// .Completed
```

SIGNAL PRODUCERS

```
extension NSNotificationCenter {
    func rac_notifications(name: String, object: AnyObject? = nil) -> SignalProducer<NSNotification, NoError> {
        return SignalProducer { observer, disposable in
            let notificationObserver = self.addObserverForName(name, object: object, queue: nil) { notification in
                observer.sendNext(notification)
            }

            disposable.addDisposable {
                self.removeObserver(notificationObserver)
            }
        }
    }
}
```

SIGNAL PRODUCERS

STARTWITH*

```
struct SignalProducer<Value, Error: ErrorType> {
  func startWithNext(next: Value -> ()) -> Disposable
  func startWithCompleted(completed: () -> ()) -> Disposable
  func startWithFailed(failed: Error -> ()) -> Disposable
  func startWithInterrupted(interrupted: () -> ()) -> Disposable
}
```

SIGNAL PRODUCERS

```
let oneTwoThreeProducer = SignalProducer<Int, NoError>(values: [1, 2, 3])
oneTwoThreeProducer.startWithNext { intValue in
    print("\(intValue)")
}
```

BUFFERS

QUEUE FOR EVENTS THAT REPLAYS THOSE EVENTS WHEN NEW SIGNALS ARE CREATED FROM THE PRODUCER.

BUFFERS

```
struct SignalProducer<Value, Error: ErrorType> {
  static func buffer(capacity: Int) ->
    (SignalProducer, Signal<Value, Error>.Observer)
}
```

SCHEDULERS

A SCHEDULER IS A SERIAL EXECUTION QUEUE TO PERFORM WORK OR DELIVER RESULTS UPON.

SCHEDULERS REPRESENTED BY

- IMMEDIATESCHEDULER
  - UISCHEDULER
- QUEUESCHEDULER
- TESTSCHEDULER

IMMEDIATESCHEDULER

A SCHEDULER THAT PERFORMS ALL WORK SYNCHRONOUSLY

UISCHEDULER

A SCHEDULER THAT PERFORMS ALL WORK ON THE MAIN THREAD

QUEUESCHEDULER

A SCHEDULER BACKED BY A SERIAL GCD QUEUE

TESTSCHEDULER

A SCHEDULER THAT IMPLEMENTS VIRTUALIZED TIME. FOR USE IN TESTING

- PROPERTIES
- ACTIONS

PROPERTIES

A PROPERTY STORES A VALUE AND NOTIFIES OBSERVERS ABOUT FUTURE CHANGES TO THAT VALUE

PROPERTIES

```
protocol PropertyType {
    associatedtype Value

    var value: Value { get }

    var producer: SignalProducer<Value, NoError> { get }
    var signal: Signal<Value, NoError> { get }
}
```

PROPERTIES REPRESENTED BY

- ANYPROPERTY
- CONSTANTPROPERTY
- MUTABLEPROPERTY
- ~~DYNAMICPROPERTY~~

ANYPROPERTY

A READ-ONLY PROPERTY THAT ALLOWS OBSERVATION OF ITS CHANGES

CONSTANTPROPERTY

A PROPERTY THAT NEVER CHANGES

MUTABLEPROPERTY

A MUTABLE PROPERTY THAT ALLOWS OBSERVATION OF ITS CHANGES

~~DYNAMICPROPERTY~~

WRAPS A DYNAMIC PROPERTY. OR ONE DEFINED IN OBJECTIVE-C.
USING KEY-VALUE CODING AND KEY-VALUE OBSERVING

ACTIONS

SIGNAL PRODUCER WRAPPER THAT PROVIDES CONVENIENT API FOR

- ITS SIGNAL PRODUCER EXECUTION STATE
- SIGNAL PRODUCER VALUES AND ERRORS

ACTIONS REPRESENTED BY

- ACTION
- ~~COCOAACTION~~

ACTIONS

```
class Action<Input, Output, Error: ErrorType> {
  var values: Signal<Output, NoError>
  var errors: Signal<Error, NoError>
  var executing: AnyProperty<Bool>
}
```

ACTIONS

```
class Action<Input, Output, Error: ErrorType> {
  init(_ execute: Input -> SignalProducer<Output, Error>)

  func apply(input: Input) -> SignalProducer<Output, ActionError<Error>>
}
```

~~COCOAACTION~~

COCOAACTION WRAPS AN ACTION FOR USE BY A GUI CONTROL (SUCH AS NSCONTROL OR UICONTROL). WITH KVO. OR WITH COCOA BINDINGS

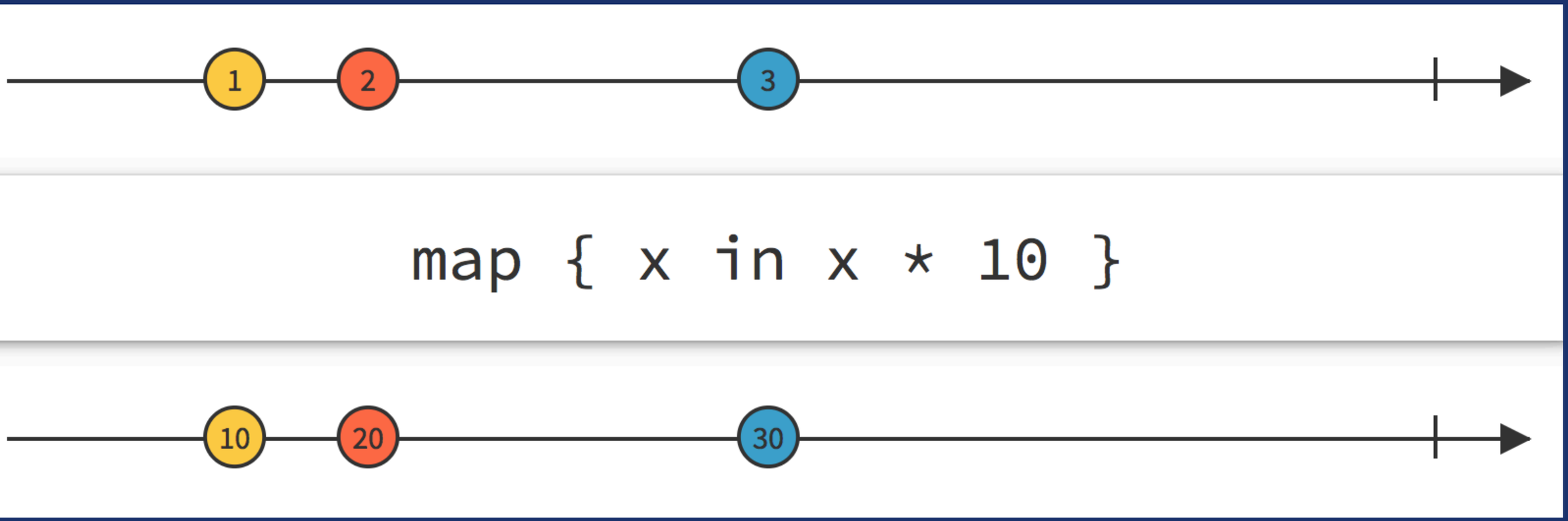
OPERATORS

OPERATORS

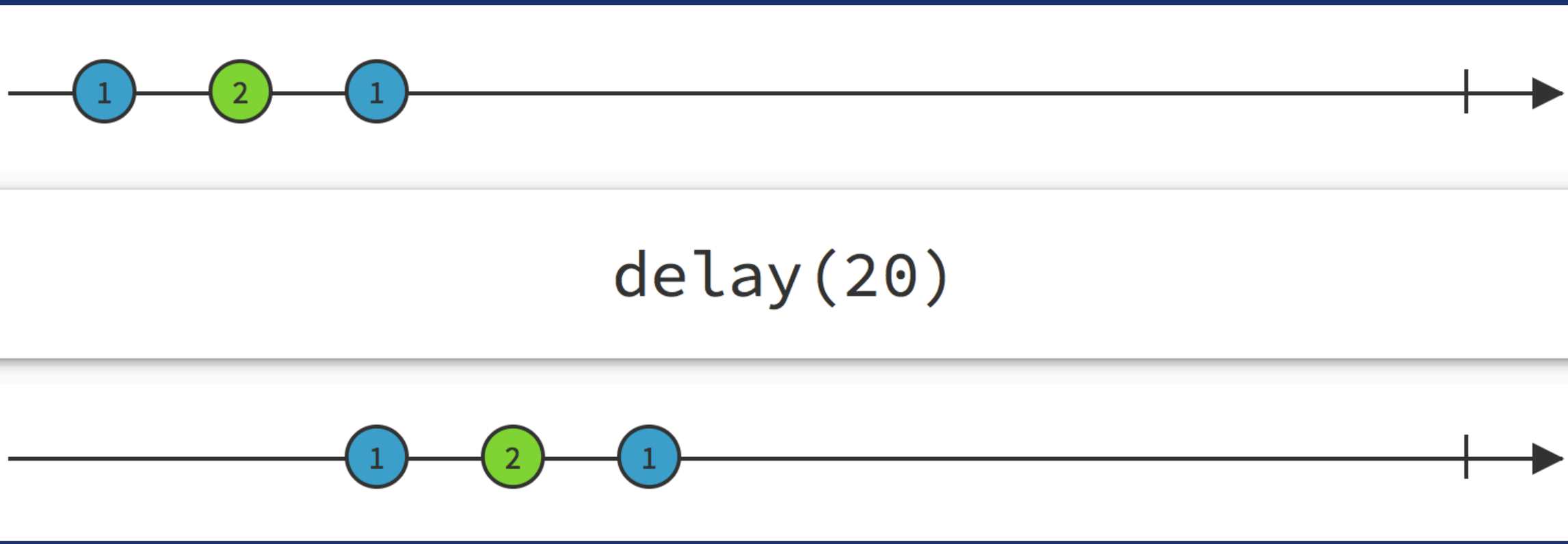
- TRANSFORMING
- FLATTENING
- AGGREGATING
- COMBINING
- FILTERING
- FAILURES

TRANSFORMING OPERATORS

MAP

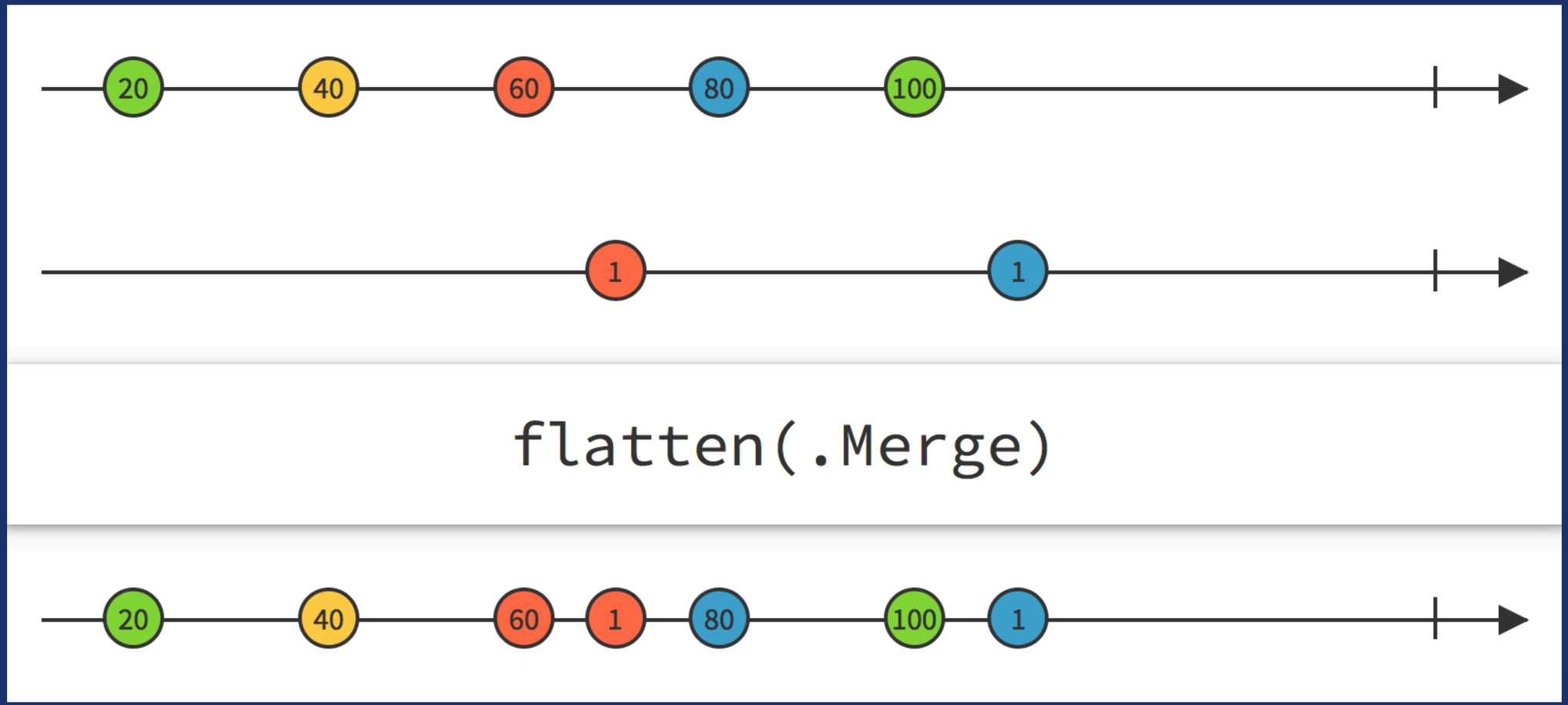


DELAY

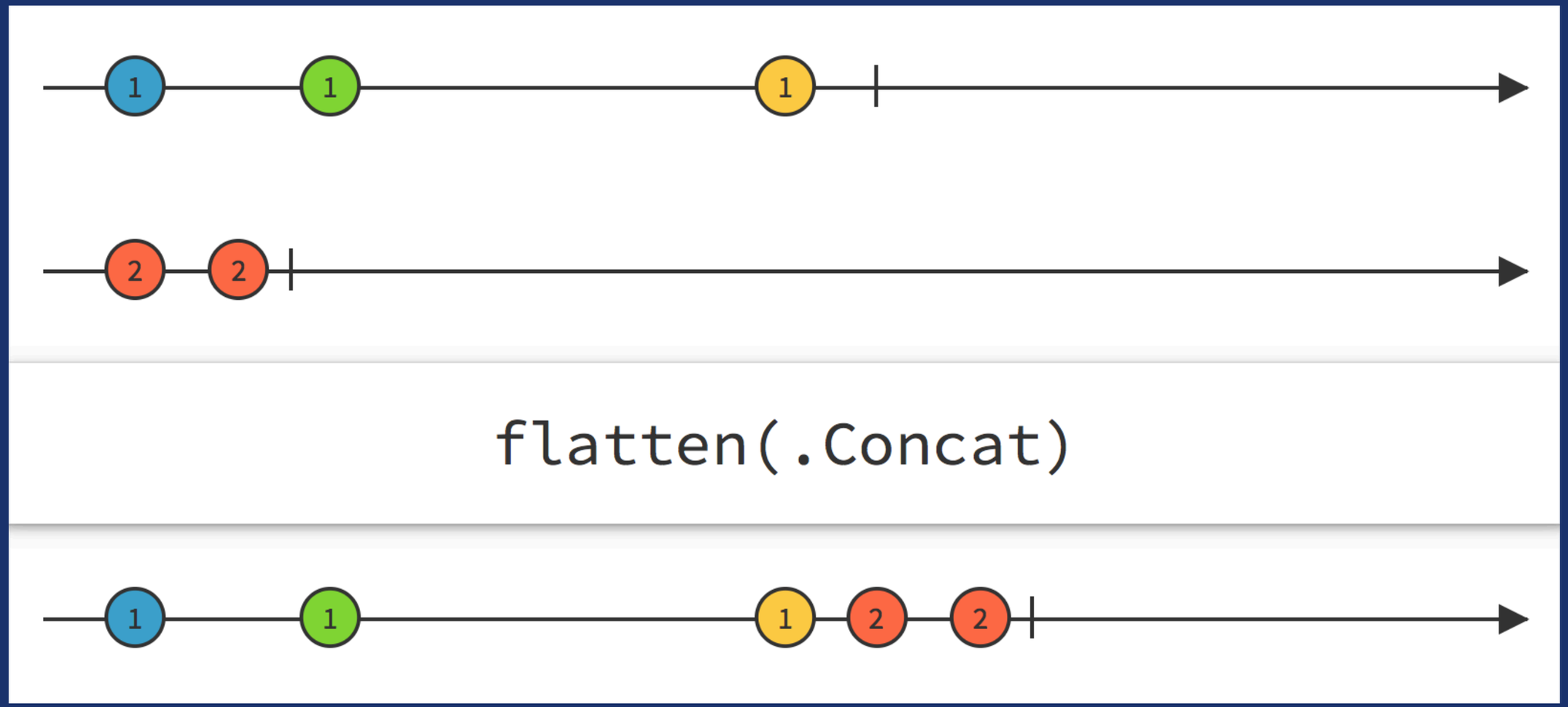


FLATTENING OPERATORS

FLATTEN(.MERGE)



FLATTEN(.CONCAT)



AGGREGATING OPERATORS

REDUCE

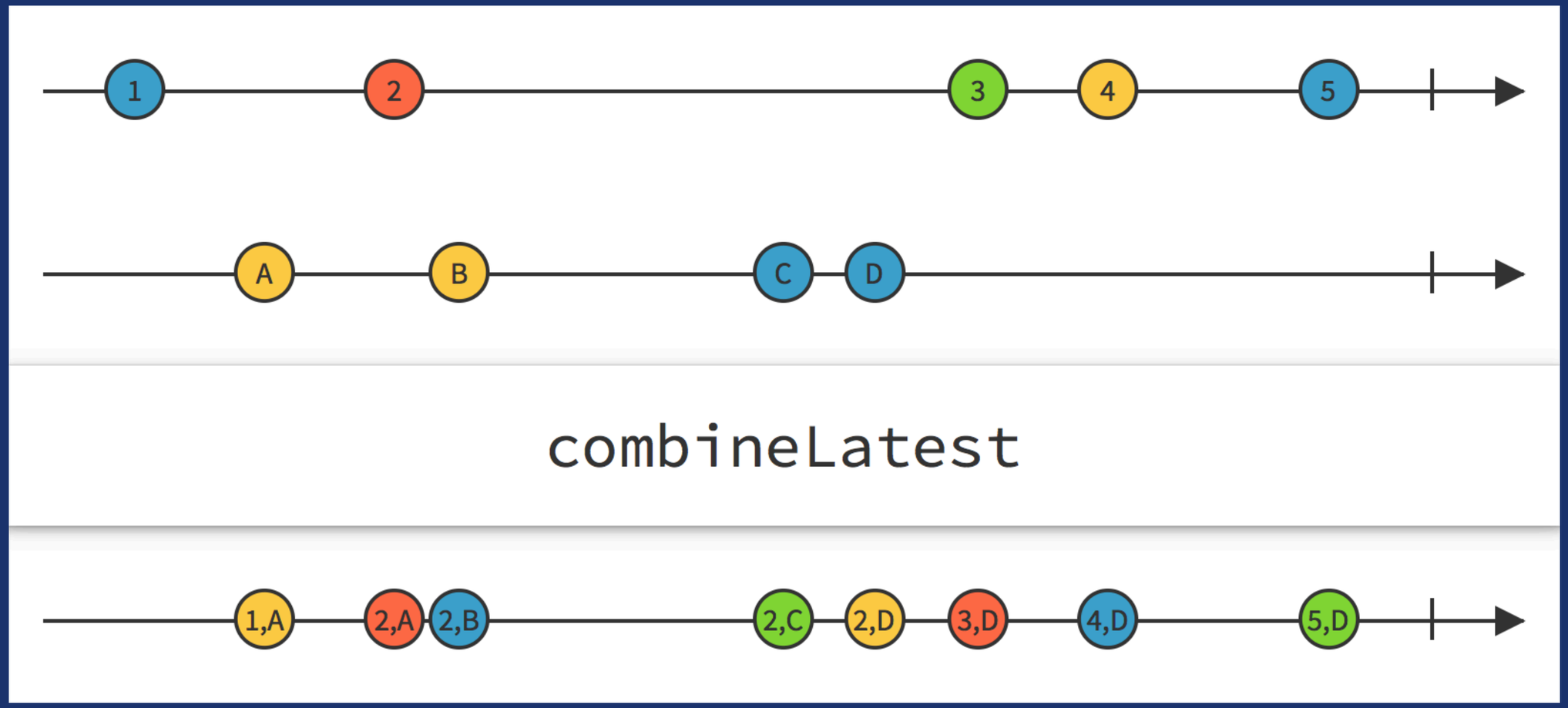

```
reduce { x, y in x + y }
```

SCAN

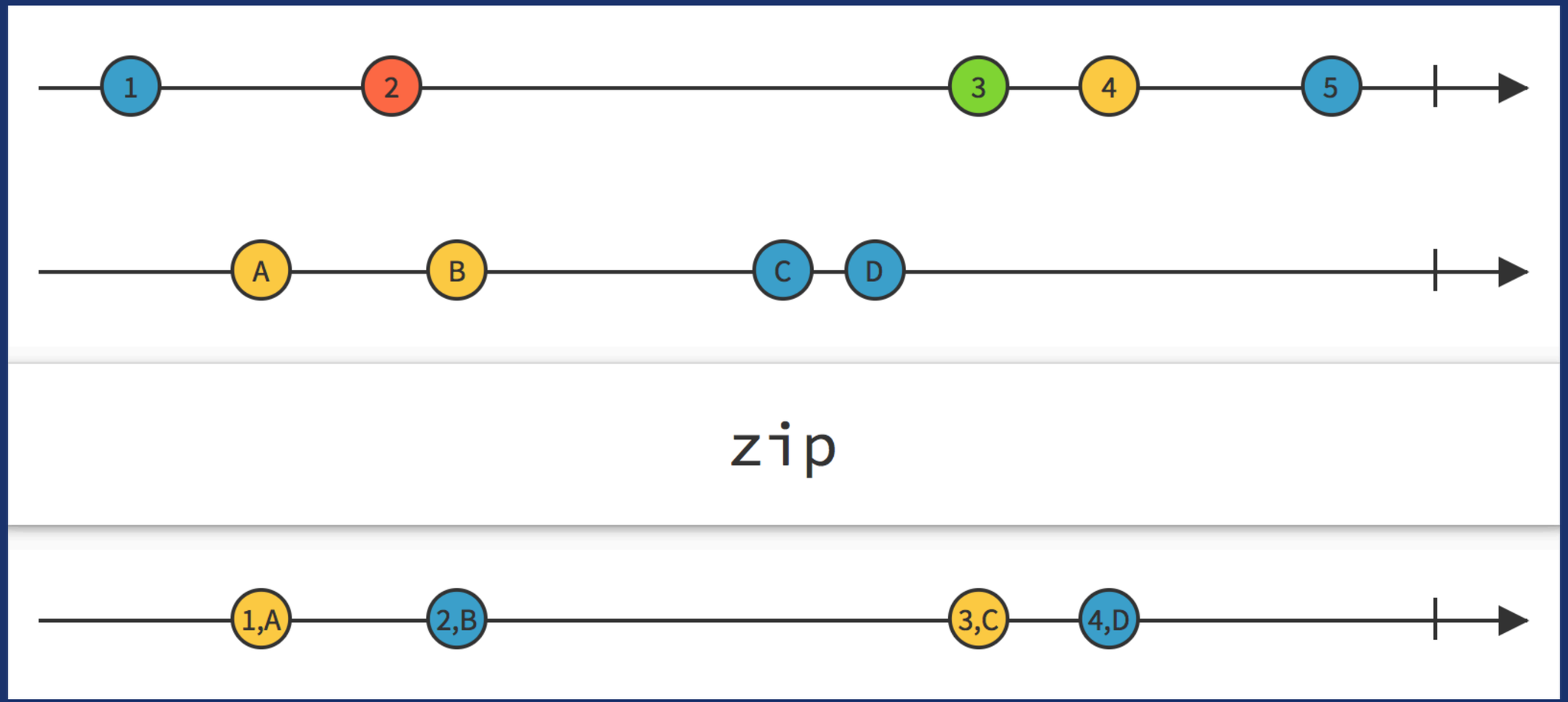

```
scan { x, y in x + y }
```

COMBINING OPERATORS

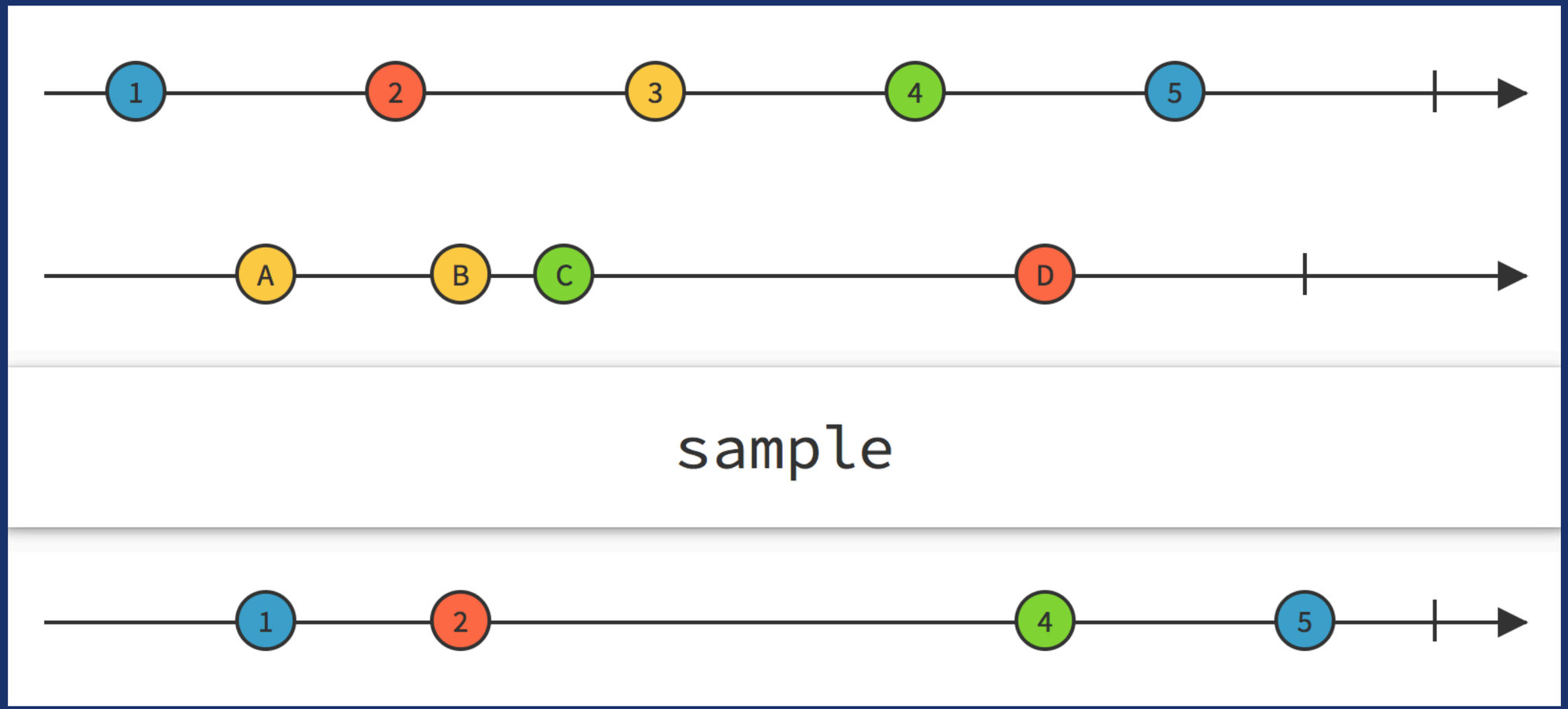
COMBINELATEST



ZIP

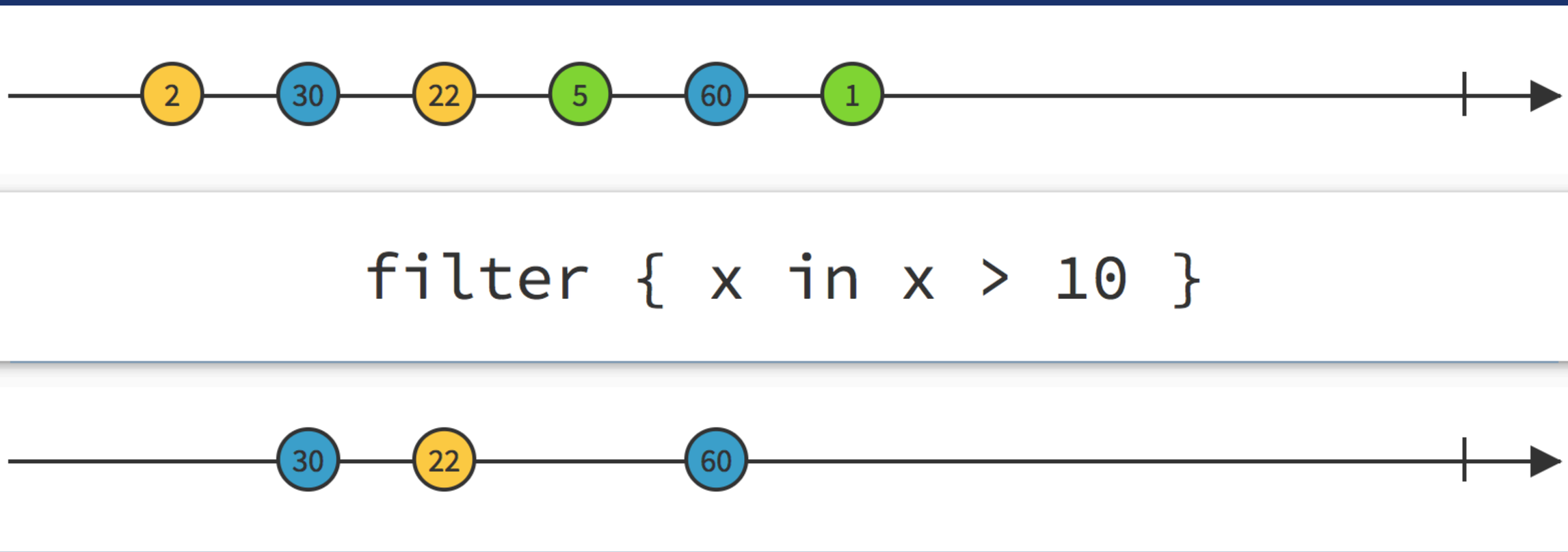


SAMPLE

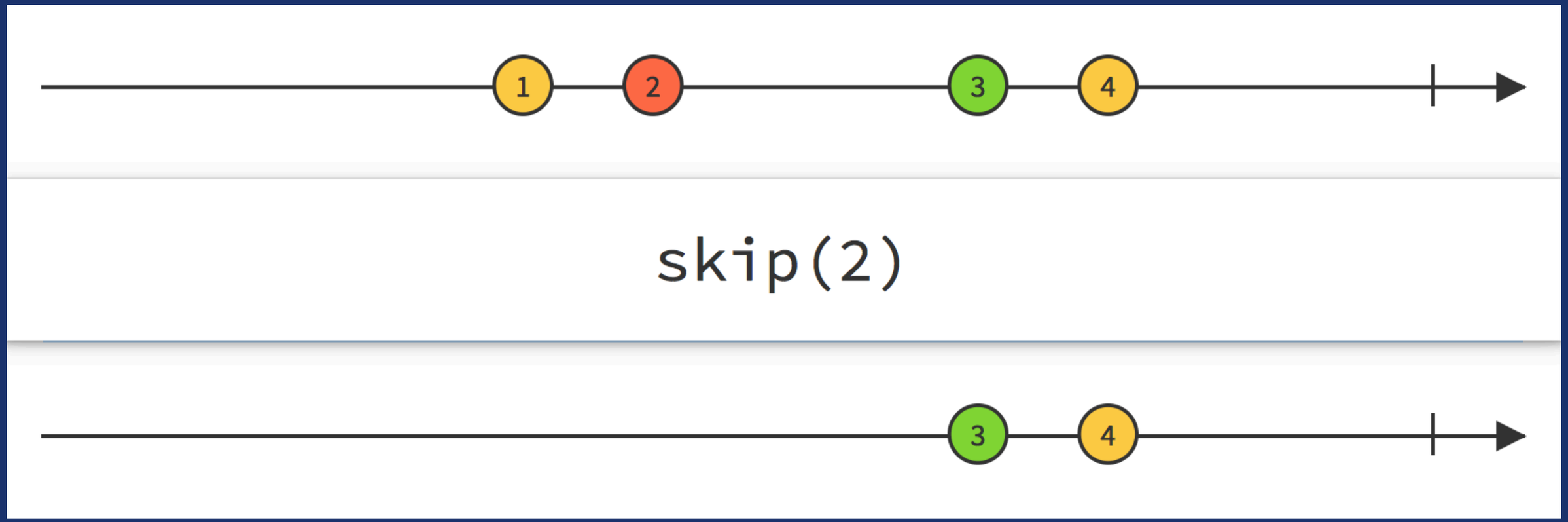


FILTERING OPERATORS

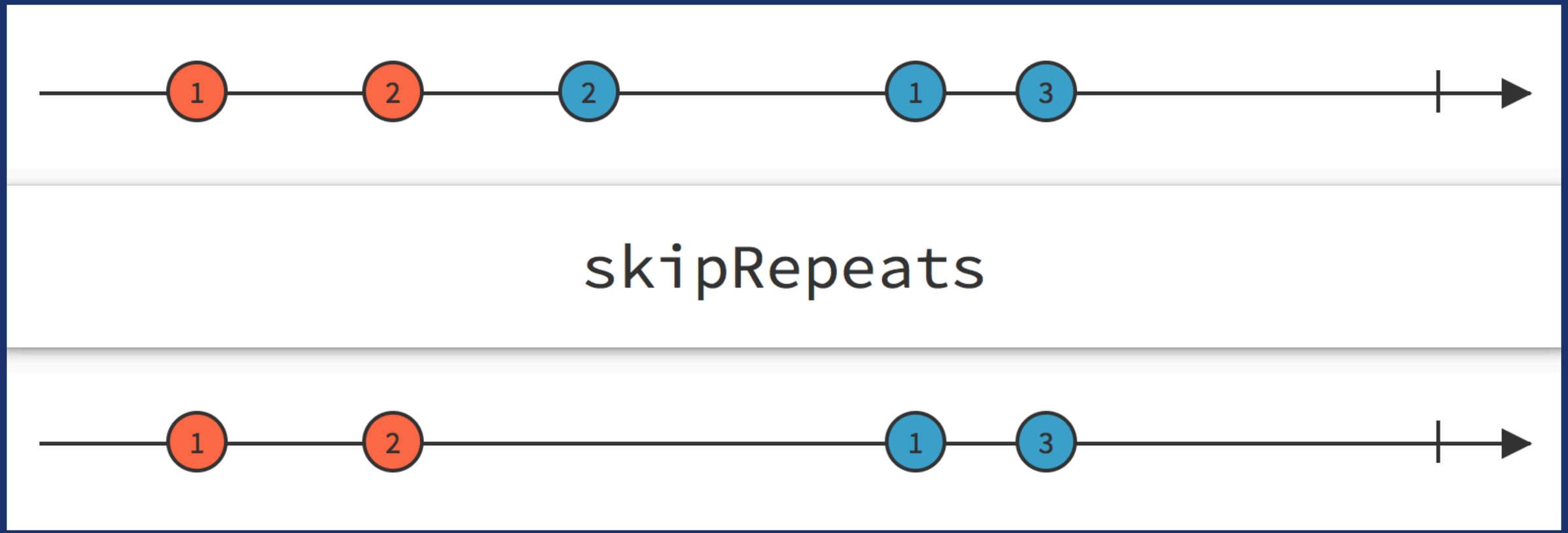
FILTER



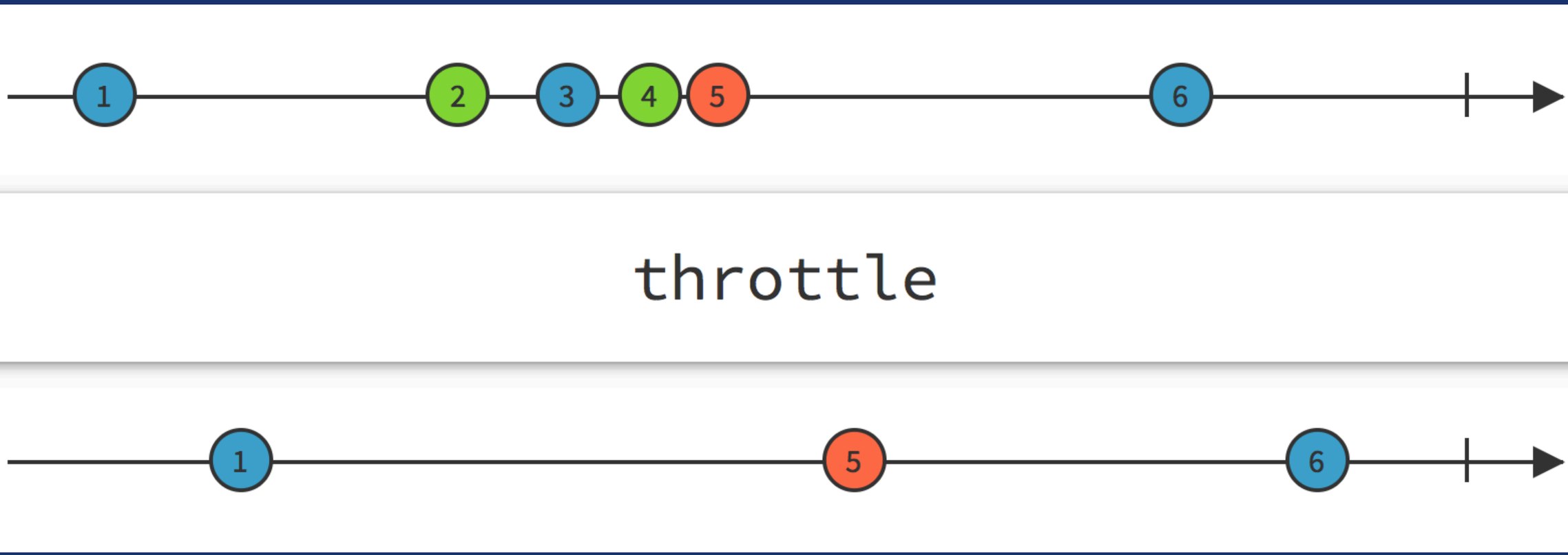
SKIP



SKIPREPEATS



THROTTLE



FAILURES OPERATORS

FLATMAPERROR

```
func flatMapError<F>(handler: Error -> SignalProducer<Value, F>) ->
  SignalProducer<Value, F>
```

RETRY

```
func retry(count: Int) ->
  SignalProducer<Value, Error>
```

EXAMPLE

```
let searchResults = searchStrings
    .flatMap(.Latest) { (query: String) -> SignalProducer<(NSData, NSURLResponse), NSError> in
        let URLRequest = self.searchRequestWithEscapedQuery(query)

        return NSURLSession.sharedSession()
            .rac_dataWithRequest(URLRequest)
            .retry(2)
            .flatMapError { error in
                print("Network error occurred: \(error)")
                return SignalProducer.empty
            }
    }
    .map { (data, URLResponse) -> String in
        let string = String(data: data, encoding: NSUTF8StringEncoding)!
        return self.parseJSONResultsFromString(string)
    }
    .observeOn(UIScheduler())
```

OFFICIAL DOCUMENTATION

- BASIC OPERATORS
- FRAMEWORK OVERVIEW

REFERENCES

- NEILPA.ME/RAC-MARBLES
- RXMARBLES.COM

END

QUESTIONS?