Observable APIs with Rx

YOW - December 2013

Ben Christensen

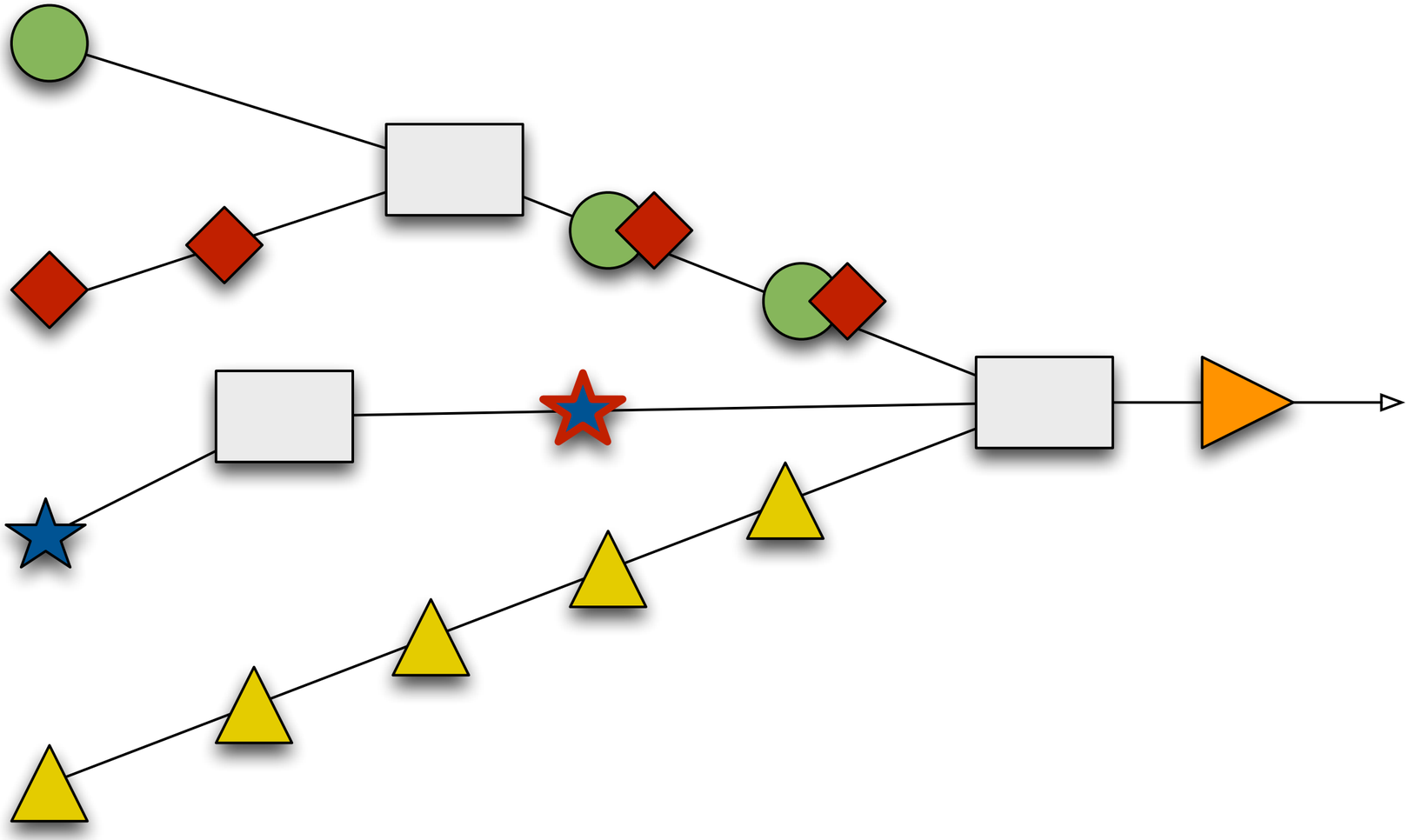
Software Engineer – Edge Platform at Netflix

@benjchristensen

http://www.linkedin.com/in/benjchristensen

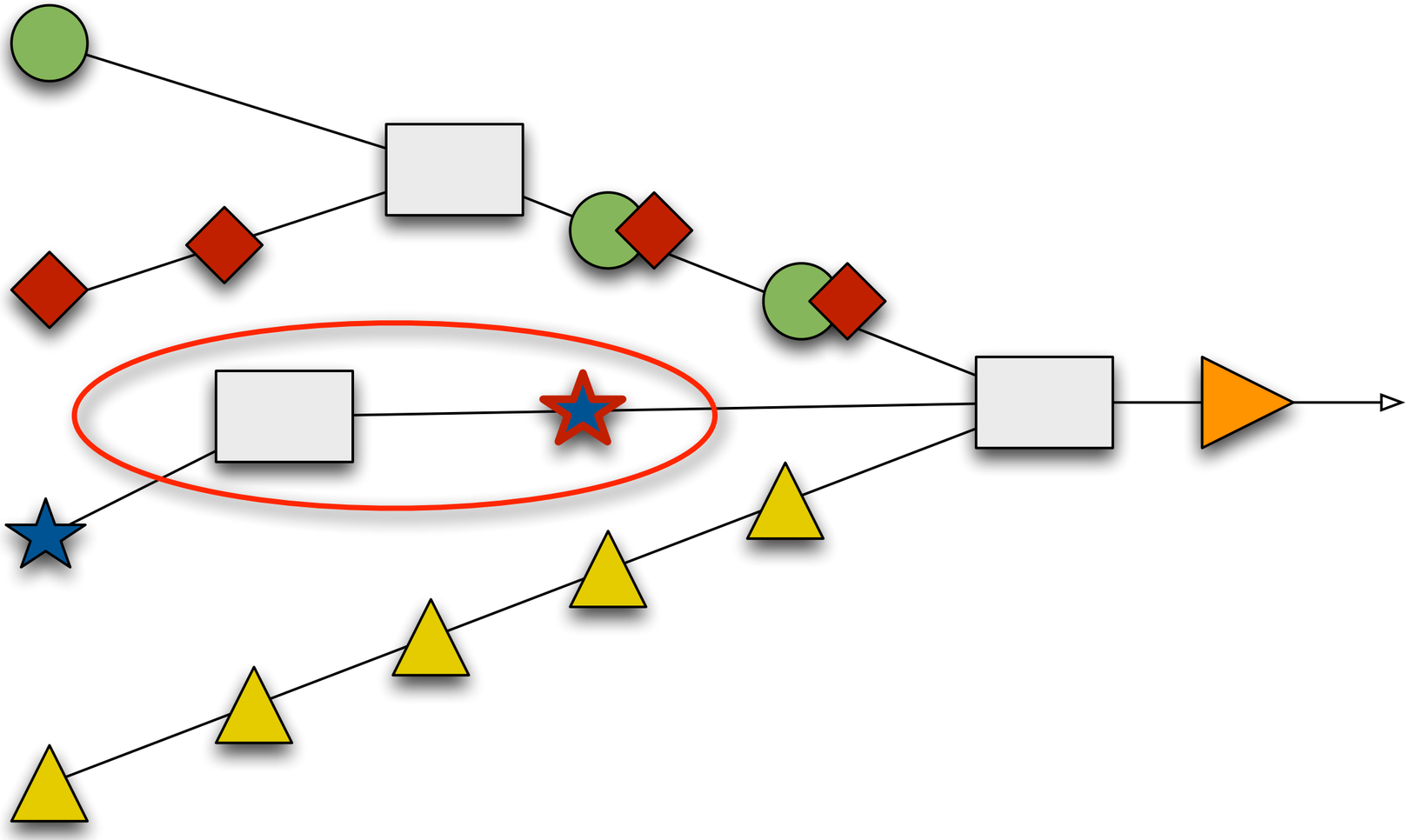
http://techblog.netflix.com/

COMPOSABLE FUNCTIONS



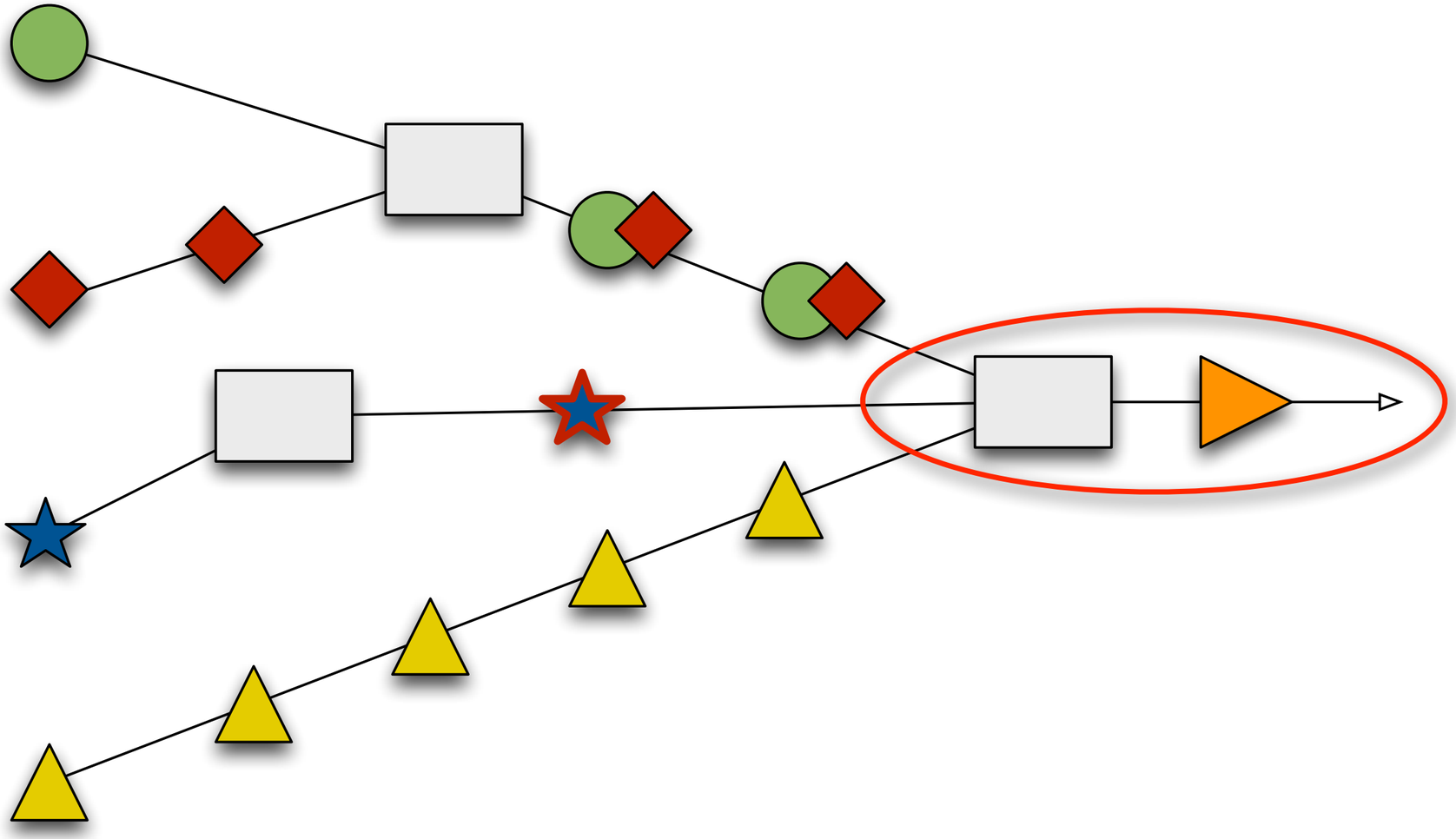
REACTIVELY APPLIED

COMPOSABLE FUNCTIONS



REACTIVELY APPLIED

COMPOSABLE FUNCTIONS



REACTIVELY APPLIED

ASYNCHRONOUS

VALUES

EVENTS

PUSH

FUNCTIONAL REACTIVE

LAMBDAS

CLOSURES

(MOSTLY) PURE

COMPOSABLE

Clojure

```
(->
  (Observable/from ["one" "two" "three"])
  (.take 2)
  (.subscribe (rx/action [arg] (println arg))))
```

Scala

```
Observable("one", "two", "three")
  .take(2)
  .subscribe((arg: String) => {
      println(arg)
  })
```

Groovy

```
Observable.from("one", "two", "three")
  .take(2)
  .subscribe({arg -> println(arg)})
```

JRuby

```
Observable.from("one", "two", "three")
  .take(2)
  .subscribe(lambda { |arg| puts arg })
```

Java8

```
Observable.from("one", "two", "three")
      .take(2)
      .subscribe((arg) -> {
            System.out.println(arg);
      });
```

Clojure

```
(->
  (Observable/from ["one" "two" "three"])
  (.take 2)
  (.subscribe (rx/action [arg] (println arg))))
```

Scala

```
Observable("one", "two", "three")
  .take(2)
  .subscribe((arg: String) => {
      println(arg)
  })
```

Groovy

```
Observable.from("one", "two", "three")
  .take(2)
  .subscribe({arg -> println(arg)})
```

JRuby

```
Observable.from("one", "two", "three")
  .take(2)
  .subscribe(lambda { |arg| puts arg })
```

Java8

```
Observable.from("one", "two", "three")
      .take(2)
      .subscribe((arg) -> {
            System.out.println(arg);
      });
```

Clojure

```
(->
  (Observable/from ["one" "two" "three"])
  (.take 2)
  (.subscribe (rx/action [arg] (println arg))))
```

Scala

```
Observable("one", "two", "three")
  .take(2)
  .subscribe((arg: String) => {
      println(arg)
  })
```

Groovy

```
Observable.from("one", "two", "three")
  .take(2)
  .subscribe({arg -> println(arg)})
```

JRuby

```
Observable.from("one", "two", "three")
  .take(2)
  .subscribe(lambda { |arg| puts arg })
```

Java8

```
Observable.from("one", "two", "three")
      .take(2)
      .subscribe((arg) -> {
            System.out.println(arg);
      });
```

RxJava

http://github.com/Netflix/RxJava

"A LIBRARY FOR COMPOSING ASYNCHRONOUS AND EVENT-BASED PROGRAMS USING OBSERVABLE SEQUENCES FOR THE JAVA VM"

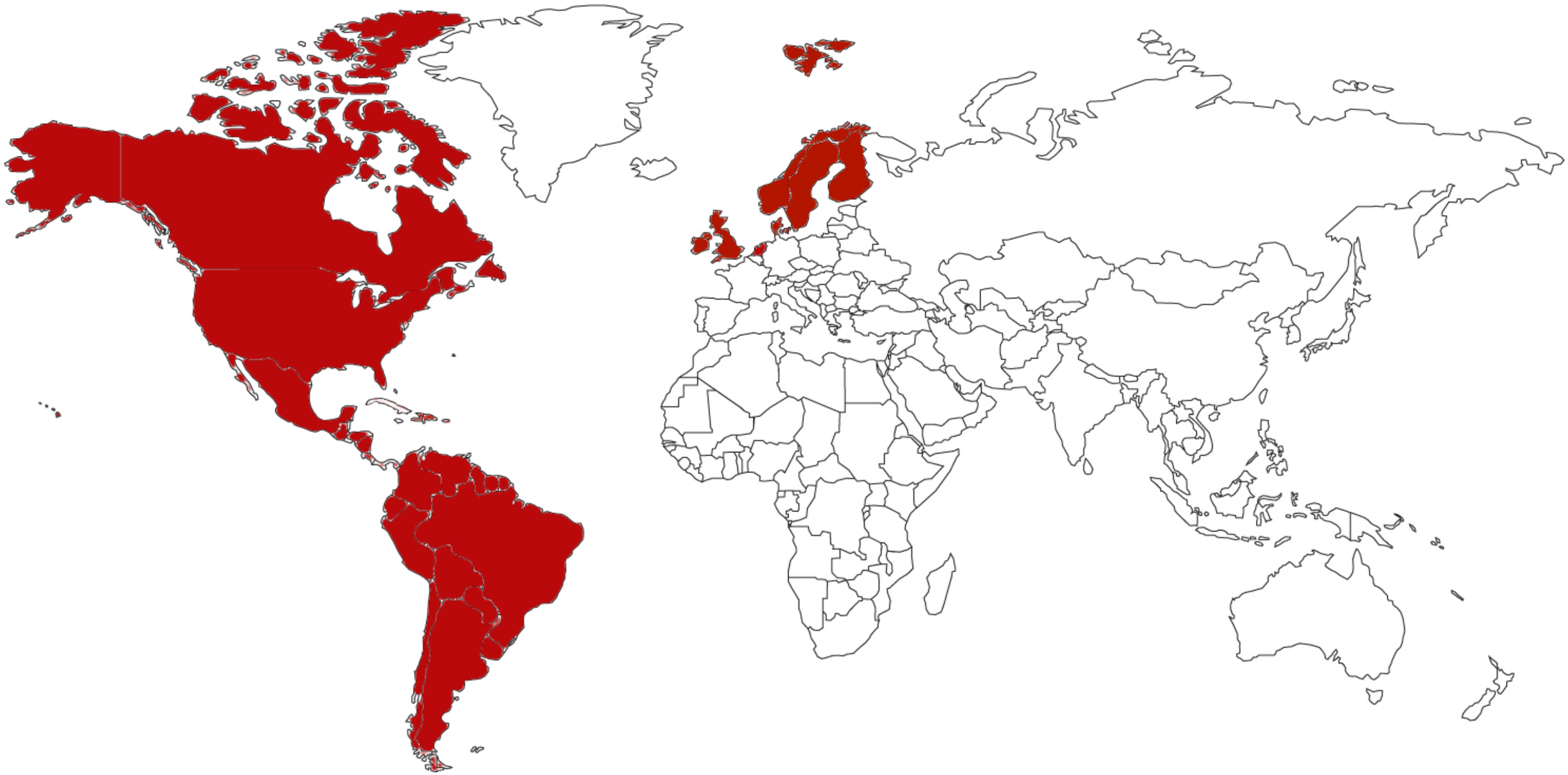
A Java port of Rx (Reactive Extensions)

https://rx.codeplex.com (.Net and Javascript by Microsoft)

NETFLIX

Sign In

More than 40 million Subscribers in 50+ Countries and Territories



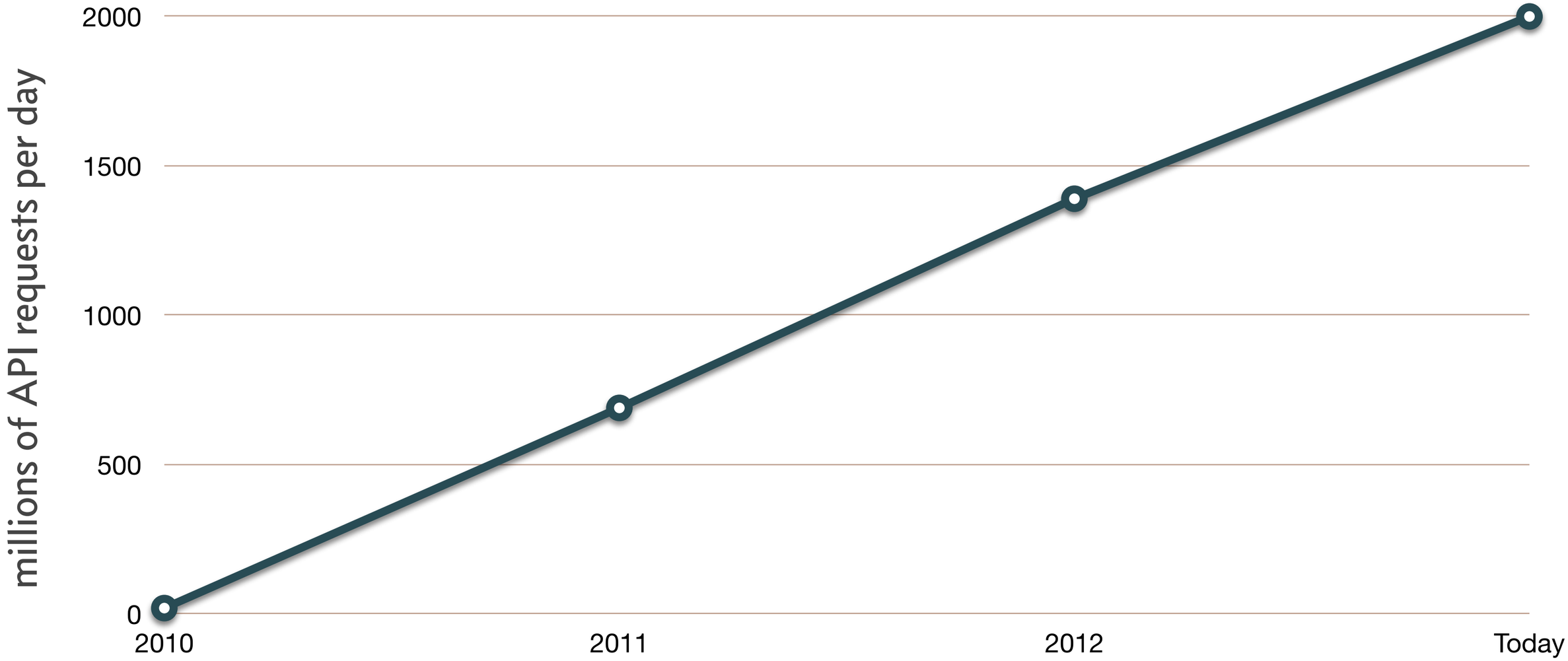
Netflix accounts for 33% of Peak Downstream Internet Traffic in North America

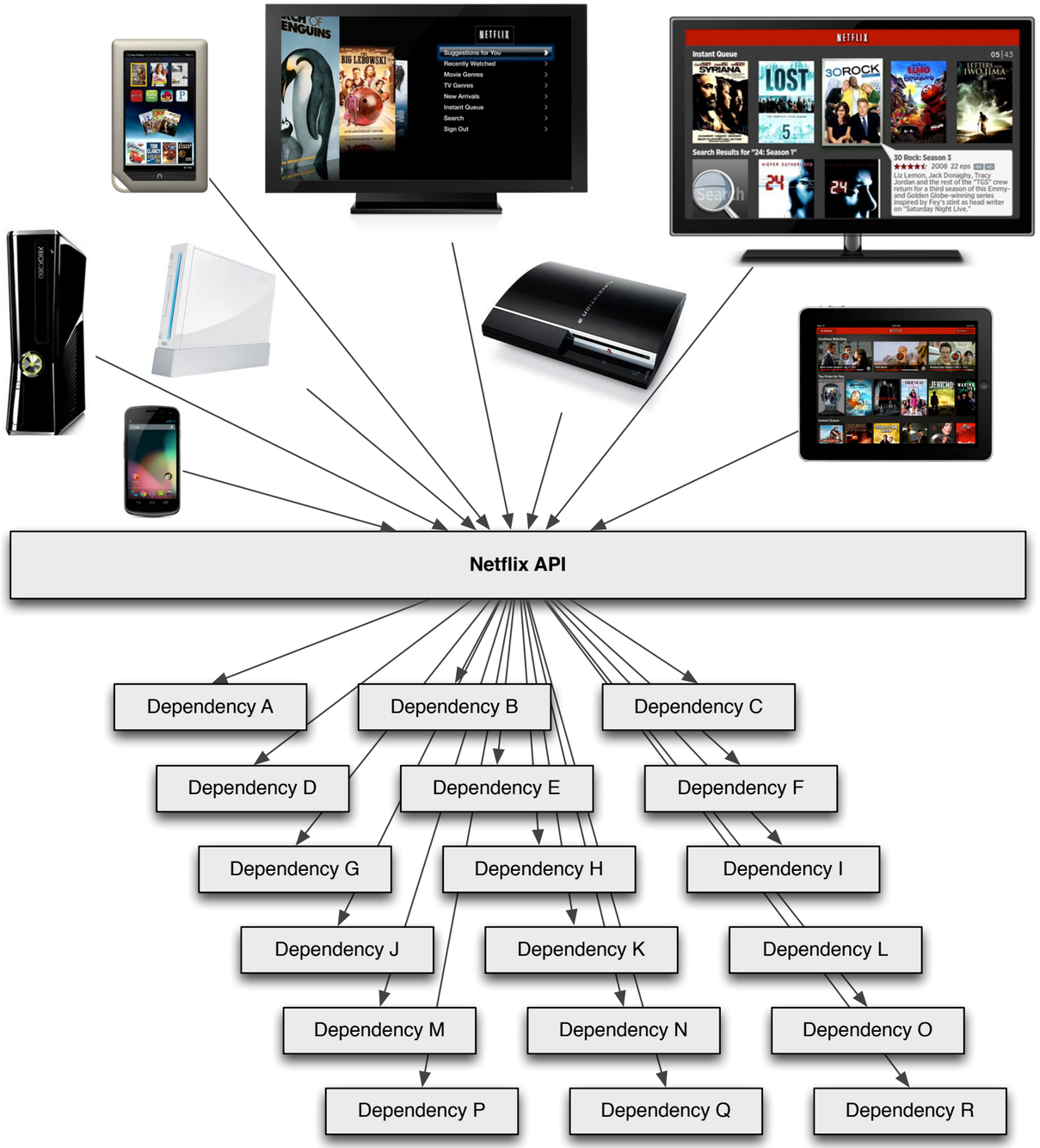
	Upstream		Downstream		Aggregate	
Rank	Application	Share	Application	Share	Application	Share
1	BitTorrent	36.8%	Netflix	33.0%	Netflix	28.8%
2	HTTP	9.83%	YouTube	14.8%	YouTube	13.1%
3	Skype	4.76%	HTTP	12.0%	HTTP	11.7%
4	Netflix	4.51%	BitTorrent	5.89%	BitTorrent	10.3%
5	SSL	3.73%	iTunes	3.92%	iTunes	3.43%
6	YouTube	2.70%	MPEG	2.22%	SSL	2.23%
7	PPStream	1.65%	Flash Video	2.21%	MPEG	2.05%
8	Facebook	1.62%	SSL	1.97%	Flash Video	2.01%
9	Apple PhotoStream	1.46%	Amazon Video	1.75%	Facebook	1.50%
10	Dropbox	1.17%	Facebook	1.48%	RTMP	1.41%
	Top 10	**68.24%**	**Top 10**	**79.01%**	**Top 10**	**76.54%**

sandvine

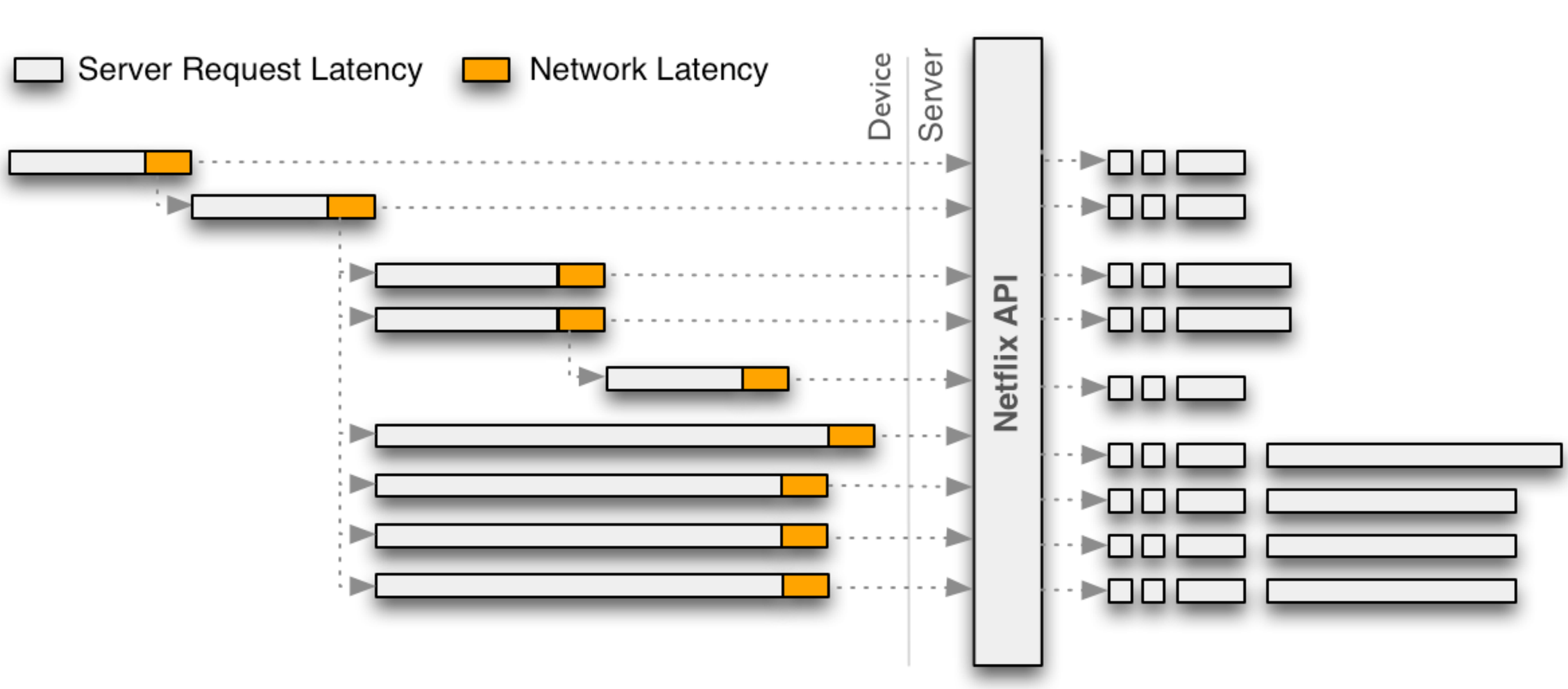
Netflix subscribers are watching more than 1 billion hours a month

API TRAFFIC HAS GROWN FROM ~20 MILLION/DAY IN 2010 TO >2 BILLION/DAY

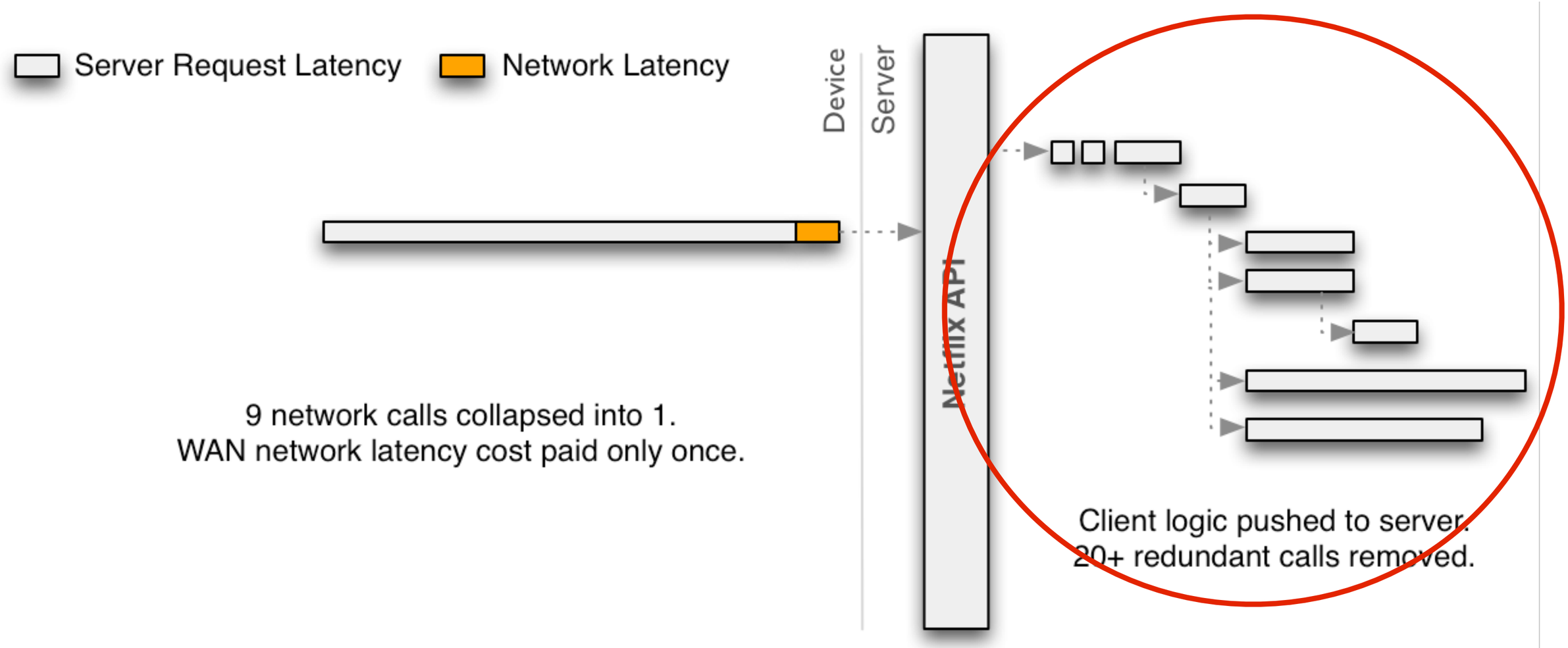



NETFLIX
Suggestions for You
Recently Watched
Movie Genres
TV Genres
New Arrivals
Instant Queue
Search
Sign Out
Instant Queue
Search Results for "24: Season 1"
30 Rock: Season 3
Netflix API
Dependency A
Dependency B
Dependency C
Dependency D
Dependency E
Dependency F
Dependency G
Dependency H
Dependency I
Dependency J
Dependency K
Dependency L
Dependency M
Dependency N
Dependency O
Dependency P
Dependency Q
Dependency R

Discovery of Rx began with a re-architecture ...

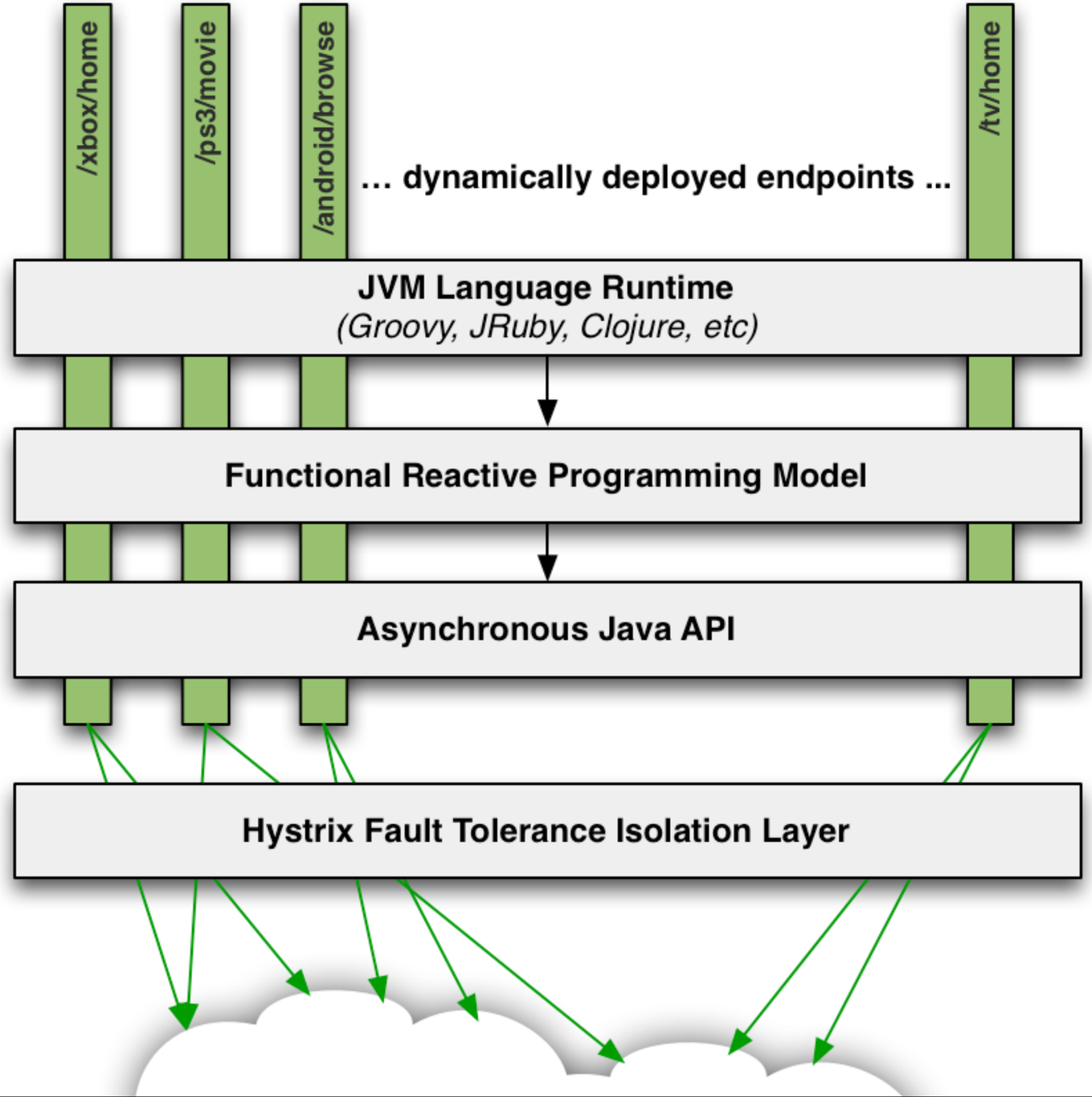


… that collapsed network traffic into coarse API calls …



NESTED, CONDITIONAL, CONCURRENT EXECUTION

… and we wanted to allow anybody to create endpoints, not just the "API Team"




Java


Clojure


Scala

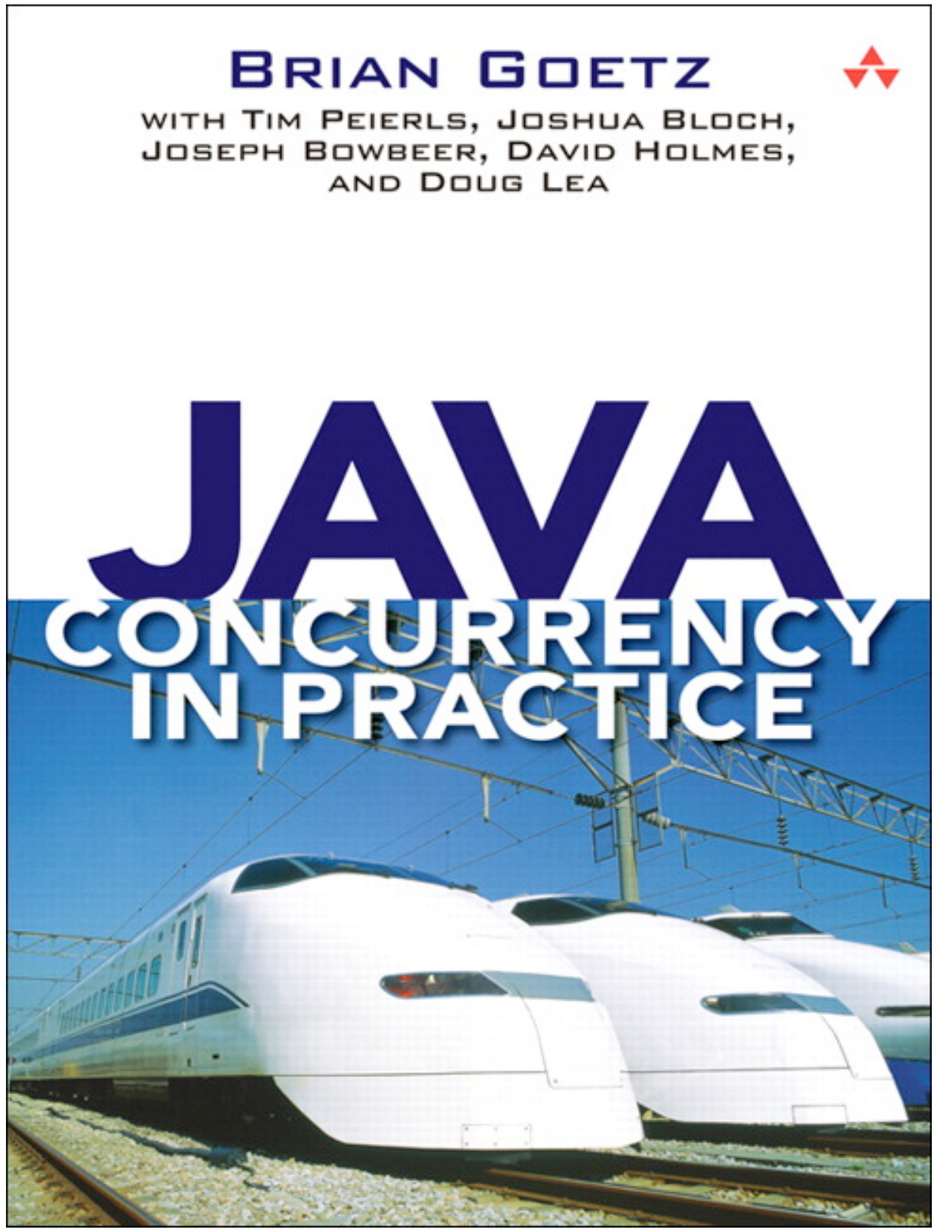

JavaScript


Groovy


JRuby

Concurrency without each engineer reading and re-reading this →

(awesome book … everybody isn't going to - or should have to - read it though, that's the point)



OWNER OF API SHOULD RETAIN CONTROL OF CONCURRENCY BEHAVIOR.

Owner of API should retain control of concurrency behavior.

```
public Data getData();
```

What if the implementation needs to change from synchronous to asynchronous?

How should the client execute that method without blocking? spawn a thread?

```
public Data getData();
```

```
public void getData(Callback<T> c);
```

```
public Future<T> getData();
```

```
public Future<List<Future<T>>> getData();
```

other options ... ?

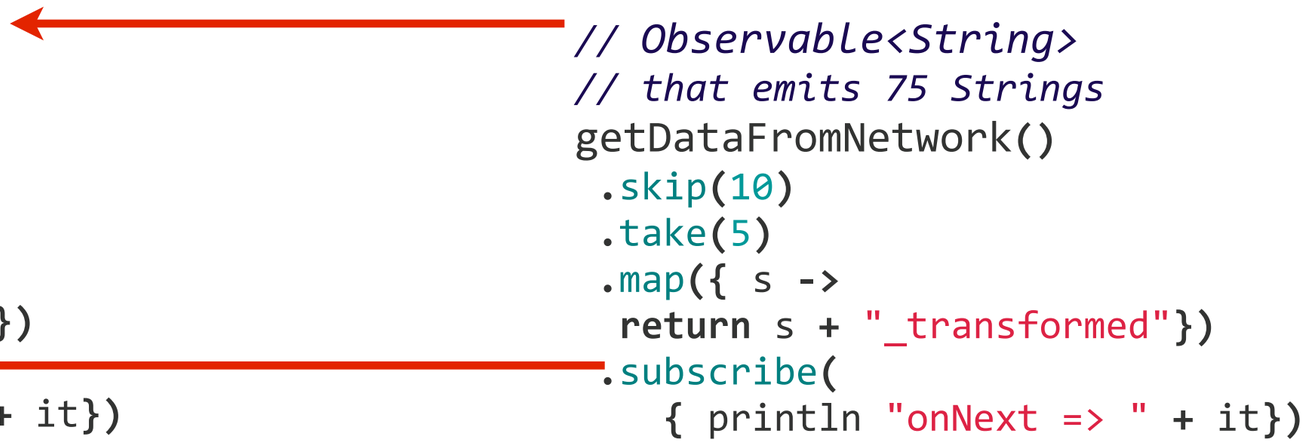
Iterable	**Observable**
pull	*push*
T next()	onNext(T)
throws Exception	onError(Exception)
returns;	onCompleted()

Iterable	Observable
pull	*push*
T next()	onNext(T)
throws Exception	onError(Exception)
returns;	onCompleted()

```
// Iterable<String>
// that contains 75 Strings
getDataFromLocalMemory()
 .skip(10)
 .take(5)
 .map({ s ->
  return s + "_transformed"})
 .forEach(
    { println "next => " + it})
```

```
// Observable<String>
// that emits 75 Strings
getDataFromNetwork()
 .skip(10)
 .take(5)
 .map({ s ->
  return s + "_transformed"})
 .subscribe(
    { println "onNext => " + it})
```

Iterable	Observable
pull	*push*
T next()	onNext(T)
throws Exception	onError(Exception)
returns;	onCompleted()



```
// Iterable<String>
// that contains 75 Strings
getDataFromLocalMemory()
 .skip(10)
 .take(5)
 .map({ s ->
  return s + "_transformed"})
 .forEach(
    { println "onNext => " + it})
```

```
// Observable<String>
// that emits 75 Strings
getDataFromNetwork()
 .skip(10)
 .take(5)
 .map({ s ->
  return s + "_transformed"})
.subscribe(
    { println "onNext => " + it})
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
String s = getData(args);
if (s.equals(x)) {
    // do something
} else {
    // do something else
}
```

	Single	Multiple
Sync	T getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
Iterable<String> values = getData(args);
for (String s : values) {
    if (s.equals(x)) {
        // do something
    } else {
        // do something else
    }
}
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
Future<String> s = getData(args);
if (s.get().equals(x)) {
    // do something
} else {
    // do something else
}
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
Future<String> s = getData(args);
if (s.get().equals(x)) {
    // do something
} else {
    // do something else
}
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
Future<String> s = getData(args);
Futures.addCallback(s,
        new FutureCallback<String> {
            public void onSuccess(String s) {
                if (s.equals(x)) {
                    // do something
                } else {
                    // do something else
                }
            }

    }, executor);
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
Future<String> s = getData(args);
Futures.addCallback(s,
        new FutureCallback<String> {
            public void onSuccess(String s) {
                if (s.equals(x)) {
                    // do something
                } else {
                    // do something else
                }
            }

    }, executor);
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
Future<String> s = getData(args);
Futures.addCallback(s,
        new FutureCallback<String> {
            public void onSuccess(String s) {
                if (s.equals(x)) {
                    // do something
                } else {
                    // do something else
                }
            }

    }, executor);
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
CompletableFuture<String> s = getData(args);
s.thenApply((v) -> {
     if (v.equals(x)) {
          // do something
     } else {
          // do something else
     }
});
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
CompletableFuture<String> s = getData(args);
s.thenApply((v) -> {
      if (v.equals(x)) {
            // do something
      } else {
            // do something else
      }
});
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
Future<String> s = getData(args);
s.map({ s ->
    if (s.equals(x)) {
        // do something
    } else {
        // do something else
    }
});
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
Future<String> s = getData(args);
s.map({ s ->
    if (s.equals(x)) {
        // do something
    } else {
        // do something else
    }
});
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
Future<String> s = getData(args);
s.map({ s ->
    if (s.equals(x)) {
        // do something
    } else {
        // do something else
    }
});
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
Observable<String> s = getData(args);
s.map({ s  ->
    if (s.equals(x)) {
        // do something
    } else {
        // do something else
    }
});
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

```
Observable<String> s = getData(args);
s.map({ s ->
    if (s.equals(x)) {
        // do something
    } else {
        // do something else
    }
});
```

	Single	Multiple
Sync	**T** getData()	**Iterable<T>** getData()
Async	**Future<T>** getData()	**Observable<T>** getData()

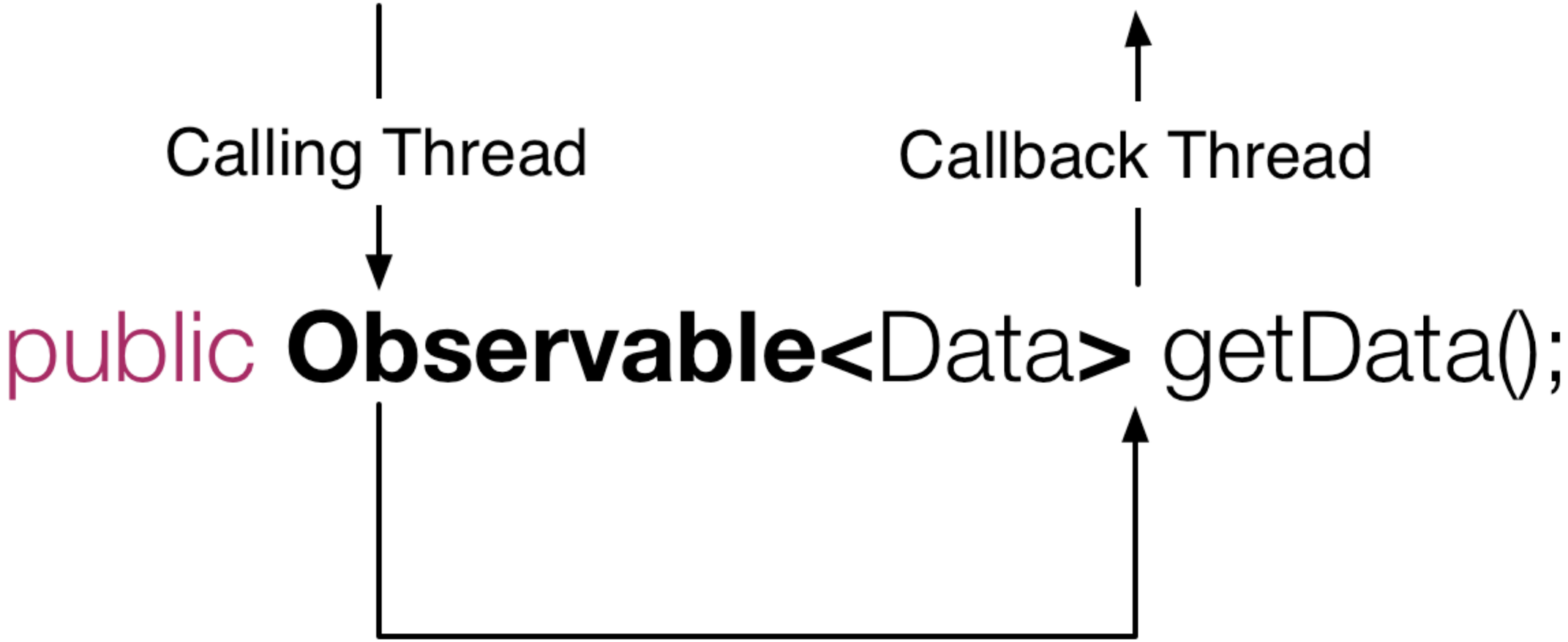
```
Observable<String> s = getData(args);
s.map({ s ->
    if (s.equals(x)) {
        // do something
    } else {
        // do something else
    }
});
```

INSTEAD OF A BLOCKING API ...

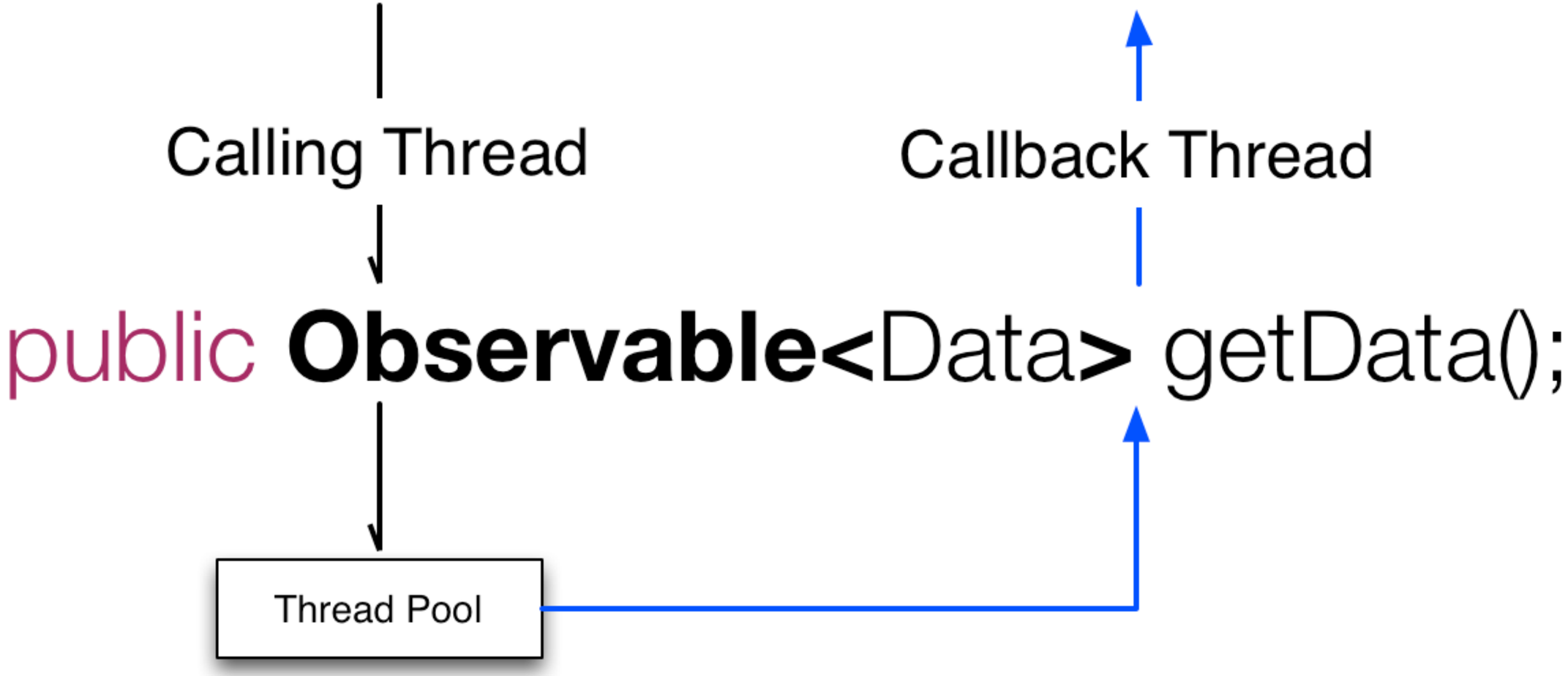
```
class VideoService {
   def VideoList getPersonalizedListOfMovies(userId);
   def VideoBookmark getBookmark(userId, videoId);
   def VideoRating getRating(userId, videoId);
   def VideoMetadata getMetadata(videoId);
}
```

... CREATE AN OBSERVABLE API:

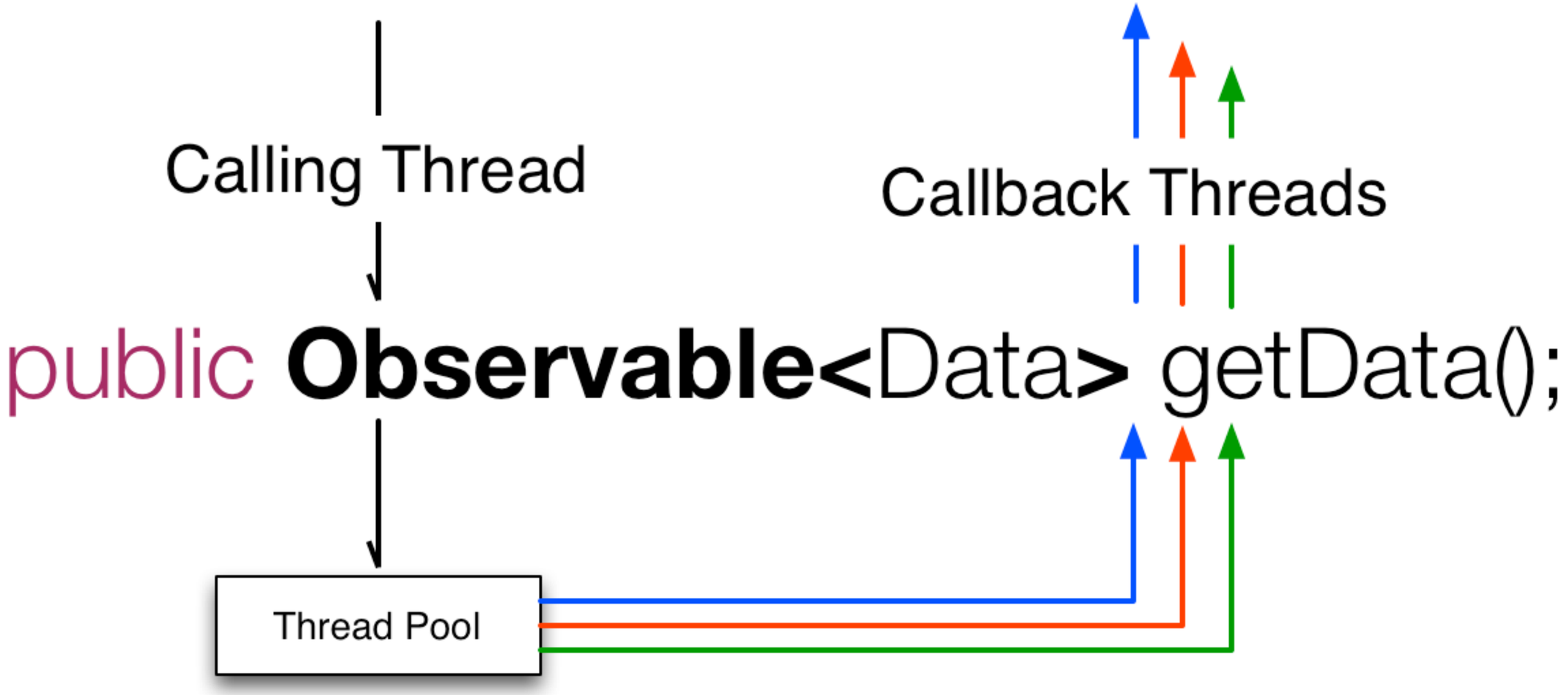
```
class VideoService {
   def Observable<VideoList> getPersonalizedListOfMovies(userId);
   def Observable<VideoBookmark> getBookmark(userId, videoId);
   def Observable<VideoRating> getRating(userId, videoId);
   def Observable<VideoMetadata> getMetadata(videoId);
}
```



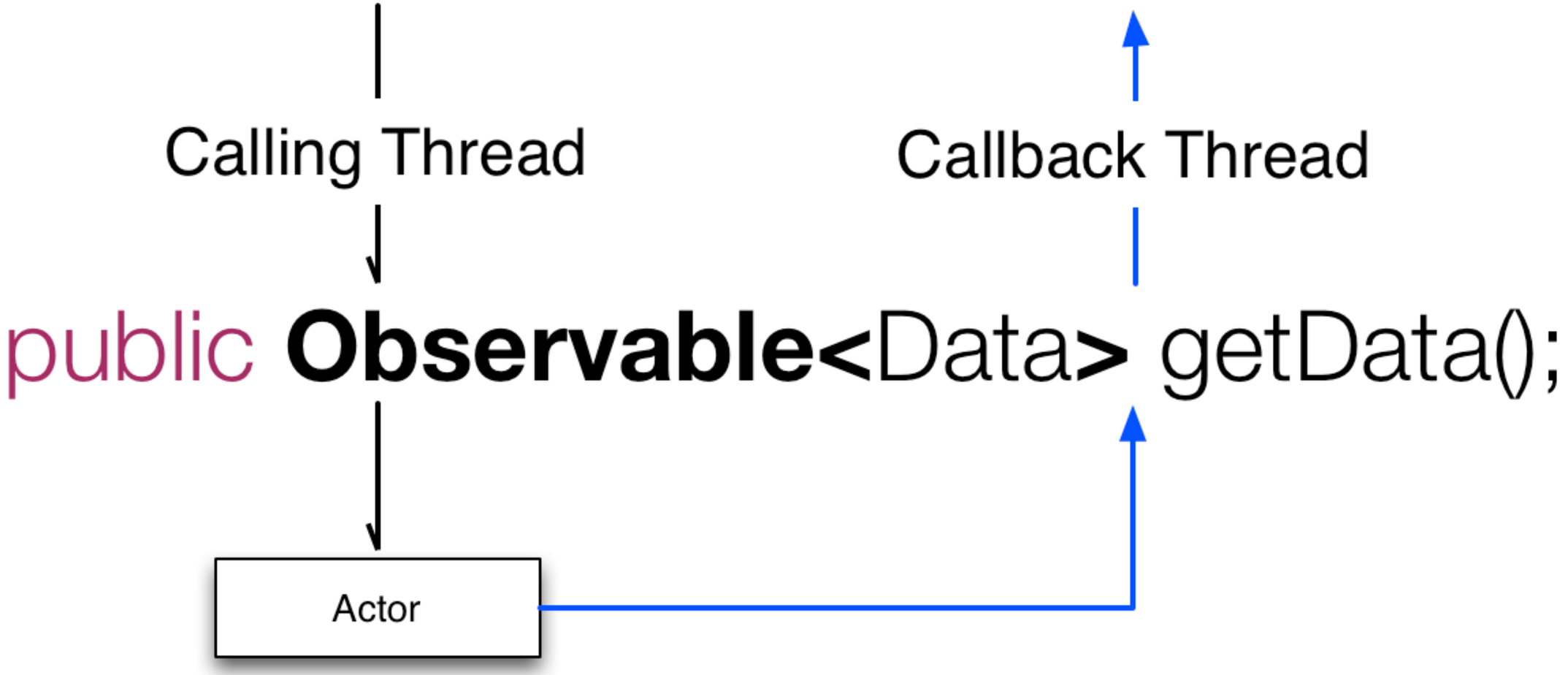
Do work synchronously on calling thread.



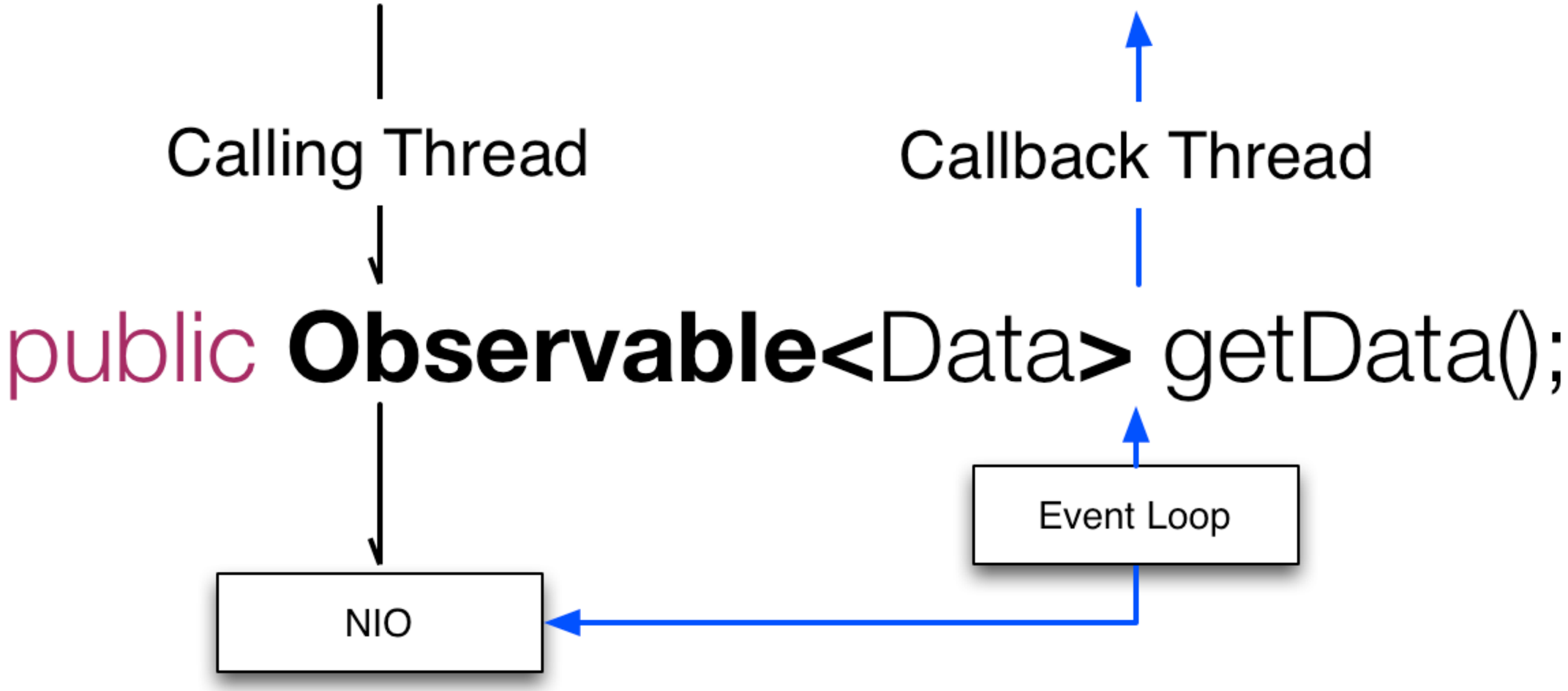
Do work asynchronously on a separate thread.



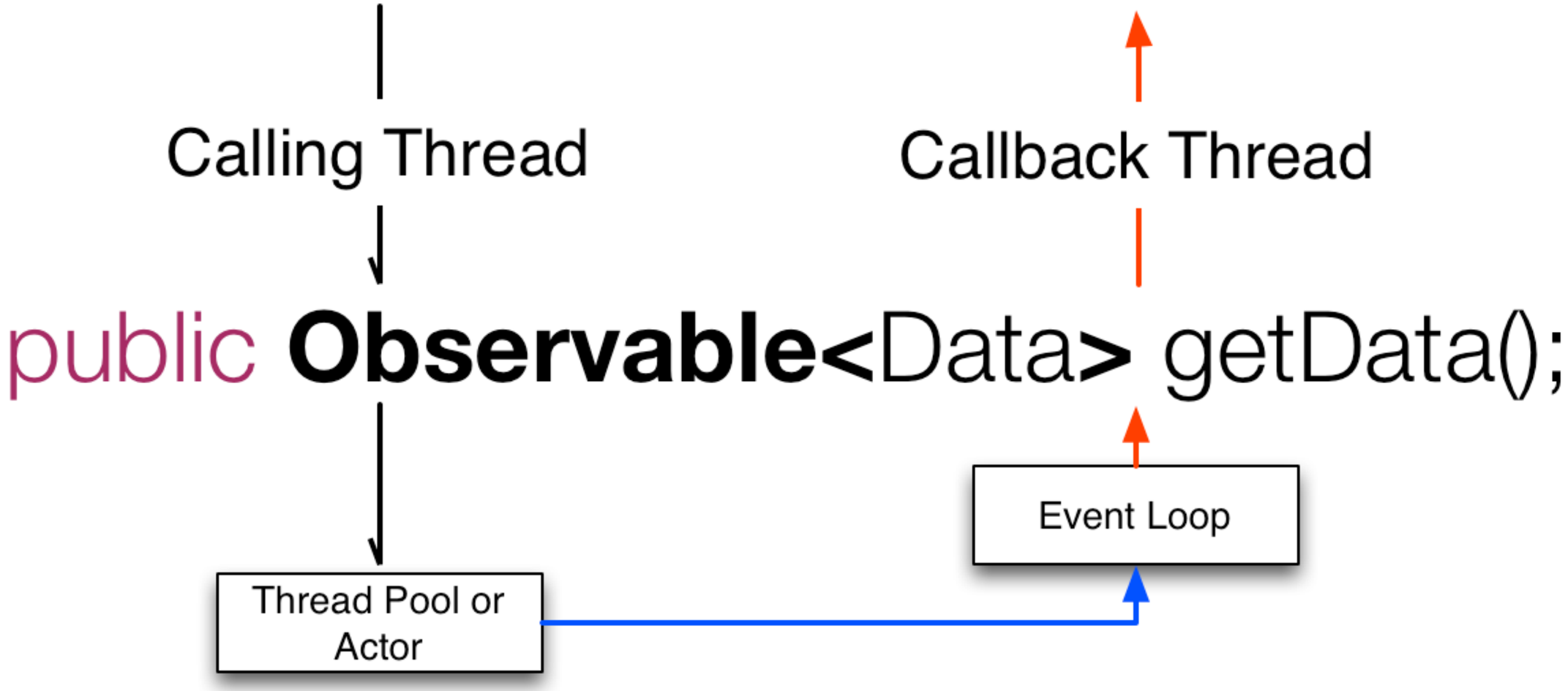
Do work asynchronously on a multiple threads.



Do work asynchronously on an actor
(or multiple actors).



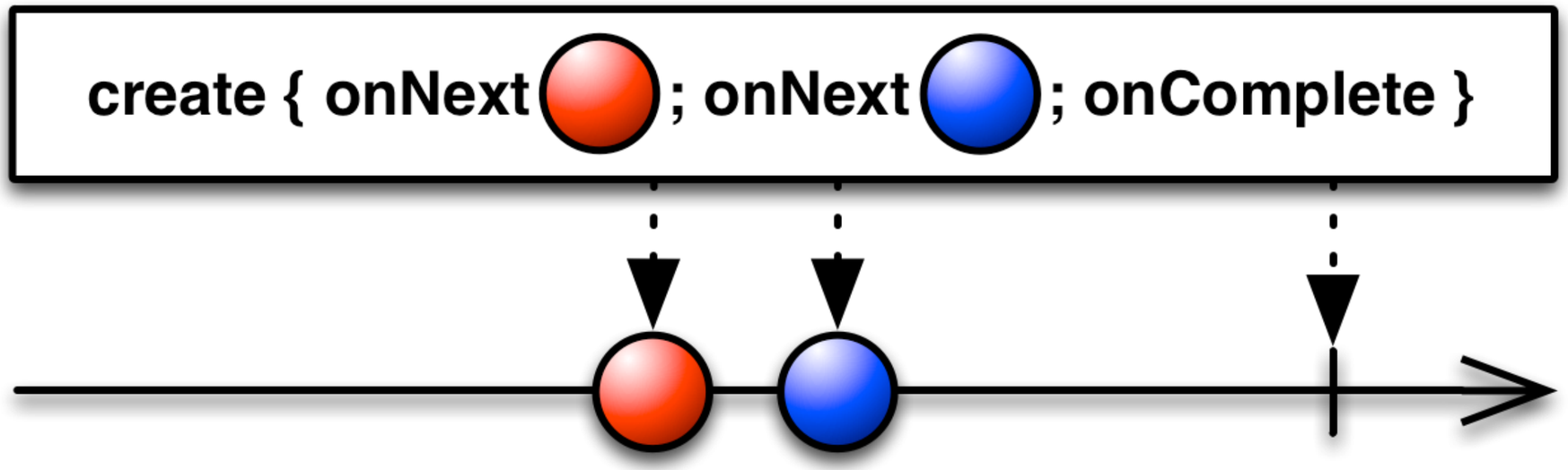
Do network access asynchronously using NIO
and perform callback on Event Loop



Do work asynchronously and perform callback via a single or multi-threaded event loop.

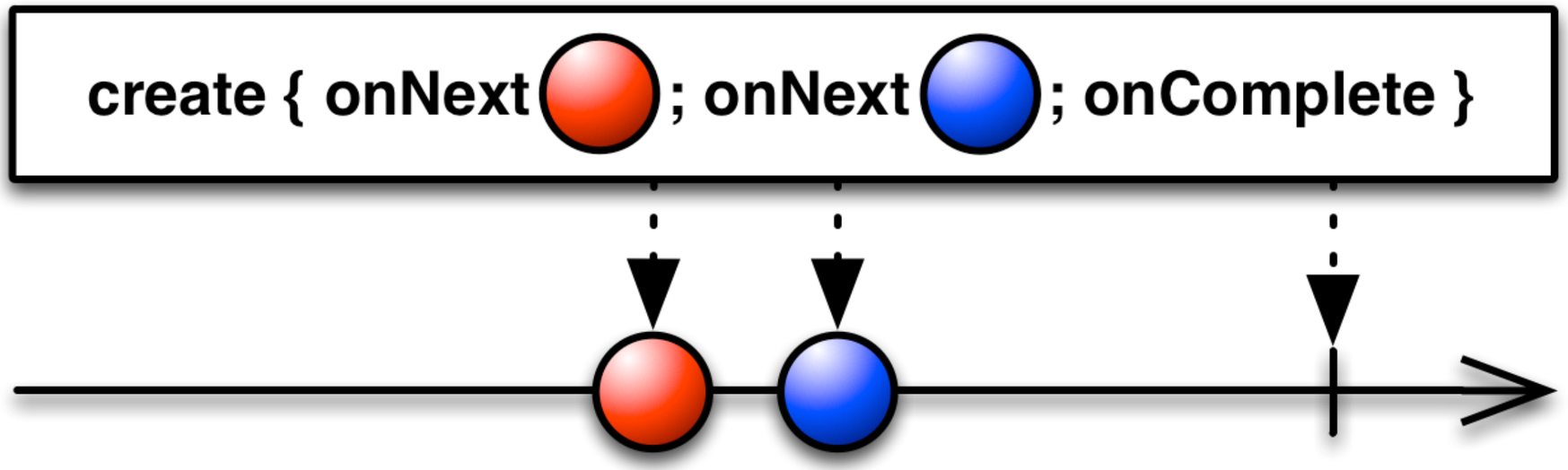
CLIENT CODE TREATS ALL INTERACTIONS WITH THE API AS ASYNCHRONOUS

THE API IMPLEMENTATION CHOOSES WHETHER SOMETHING IS BLOCKING OR NON-BLOCKING AND WHAT RESOURCES IT USES



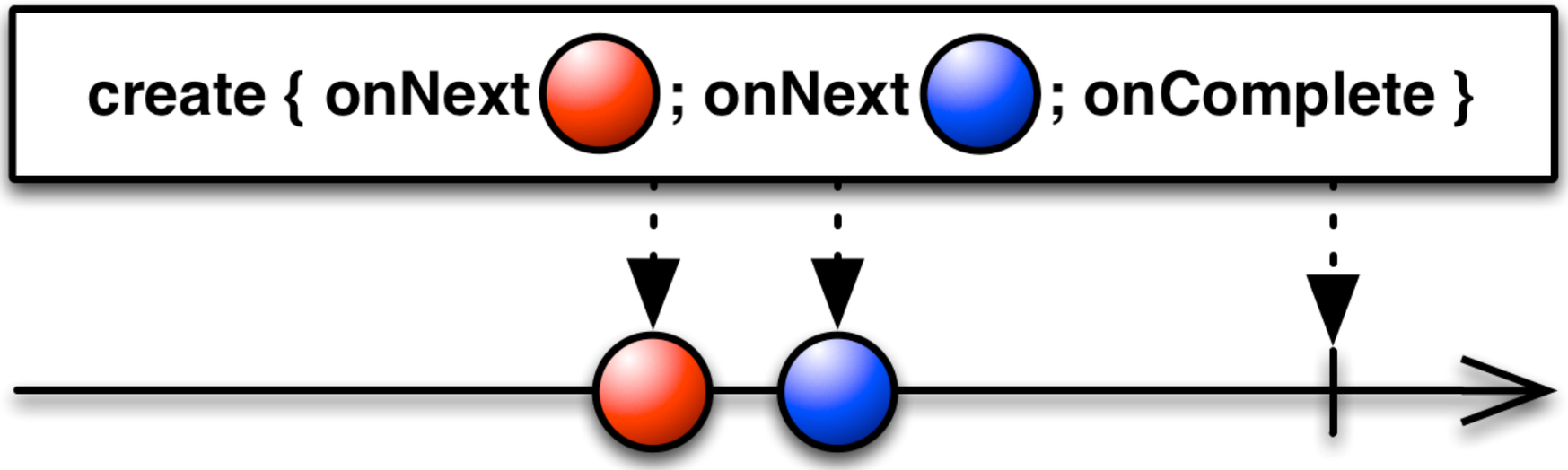

```
Observable<T> create(Func1<Observer<T>, Subscription> func)
```

```
Observable.create({ observer ->
  try {
    observer.onNext(new Video(id))
    observer.onCompleted();
  } catch(Exception e) {
    observer.onError(e);
  }
})
```



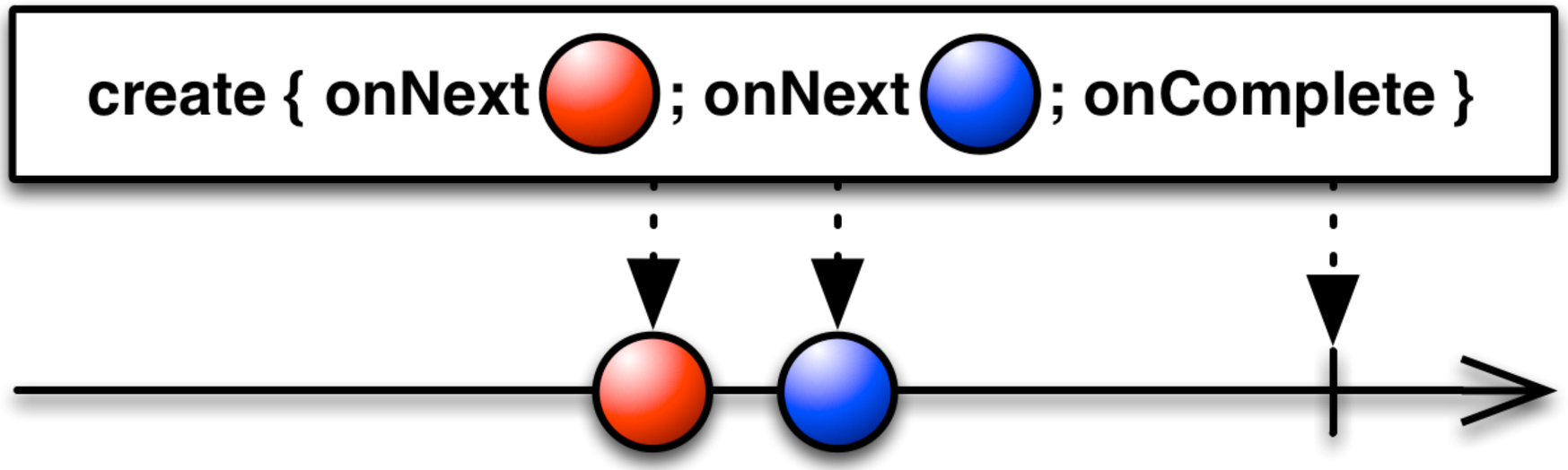

```
Observable<T> create(Func1<Observer<T>, Subscription> func)
```

```
Observable.create({ observer ->
  try {
    observer.onNext(new Video(id))
    observer.onCompleted();
  } catch(Exception e) {
    observer.onError(e);
  }
})
```



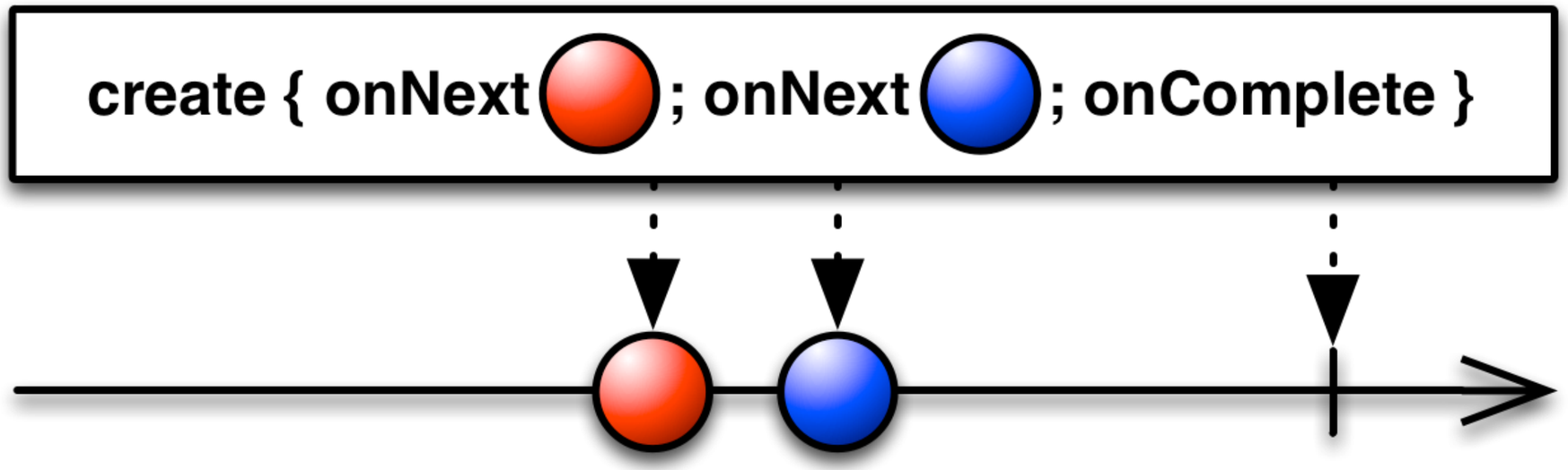

```
Observable<T> create(Func1<Observer<T>, Subscription> func)
```

```
Observable.create({ observer ->
  try {
    observer.onNext(new Video(id))
    observer.onCompleted();
  } catch(Exception e) {
    observer.onError(e);
  }
})
```



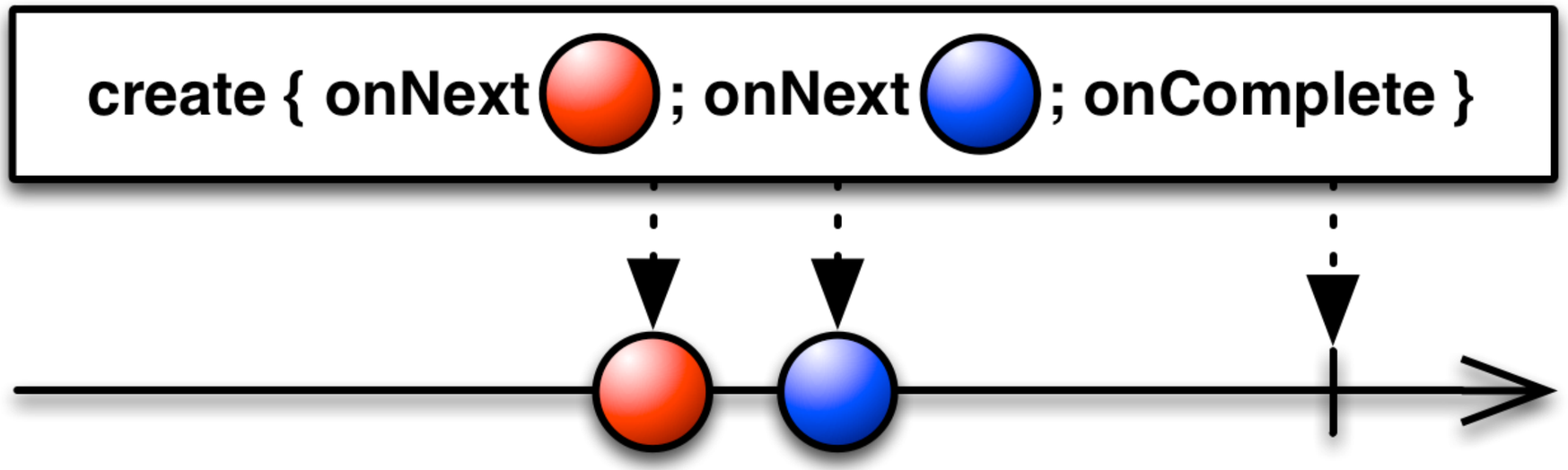

```
Observable<T> create(Func1<Observer<T>, Subscription> func)
```

```
Observable.create({ observer ->
  try {
    observer.onNext(new Video(id))
    observer.onCompleted();
  } catch(Exception e) {
    observer.onError(e);
  }
})
```




```
Observable<T> create(Func1<Observer<T>, Subscription> func)
```

```
Observable.create({ observer ->
  try {
    observer.onNext(new Video(id))
    observer.onCompleted();
  } catch(Exception e) {
    observer.onError(e);
  }
})
```




```
Observable<T> create(Func1<Observer<T>, Subscription> func)
```

```
Observable.create({ observer ->
  try {
    observer.onNext(new Video(id))
    observer.onCompleted();
  } catch(Exception e) {
    observer.onError(e);
  }
})
```

Asynchronous Observable with Single Value

```
def Observable<VideoRating> getRating(userId, videoId) {
    // fetch the VideoRating for this user asynchronously
    return Observable.create({ observer ->
        executor.execute(new Runnable() {
            def void run() {
              try {
                VideoRating rating = ... do network call ...
                observer.onNext(rating)
                observer.onCompleted();
              } catch(Exception e) {
                observer.onError(e);
              }
            }
        })
    })
}
```

Synchronous Observable with Multiple Values

```
def Observable<Video> getVideos() {
    return Observable.create({ observer ->
        try {
          for(v in videos) {
             observer.onNext(v)
          }
          observer.onCompleted();
        } catch(Exception e) {
          observer.onError(e);
        }
    })
}
```

Caution: This example is eager and will *always* emit all values regardless of subsequent operators such as take(10)

Asynchronous Observable with Multiple Values

```
def Observable<Video> getVideos() {
    return Observable.create({ observer ->
        executor.execute(new Runnable() {
            def void run() {
              try {
                for(id in videoIds) {
                  Video v = ... do network call ...
                  observer.onNext(v)
                }
                observer.onCompleted();
              } catch(Exception e) {
                observer.onError(e);
              }
            }
        })
    })
}
```

Asynchronous Observer

```
getVideos().subscribe(new Observer<Video>() {

    def void onNext(Video video) {
        println("Video: " + video.videoId)
    }

    def void onError(Exception e) {
        println("Error")
        e.printStackTrace()
    }

    def void onCompleted() {
        println("Completed")
    }
})
```

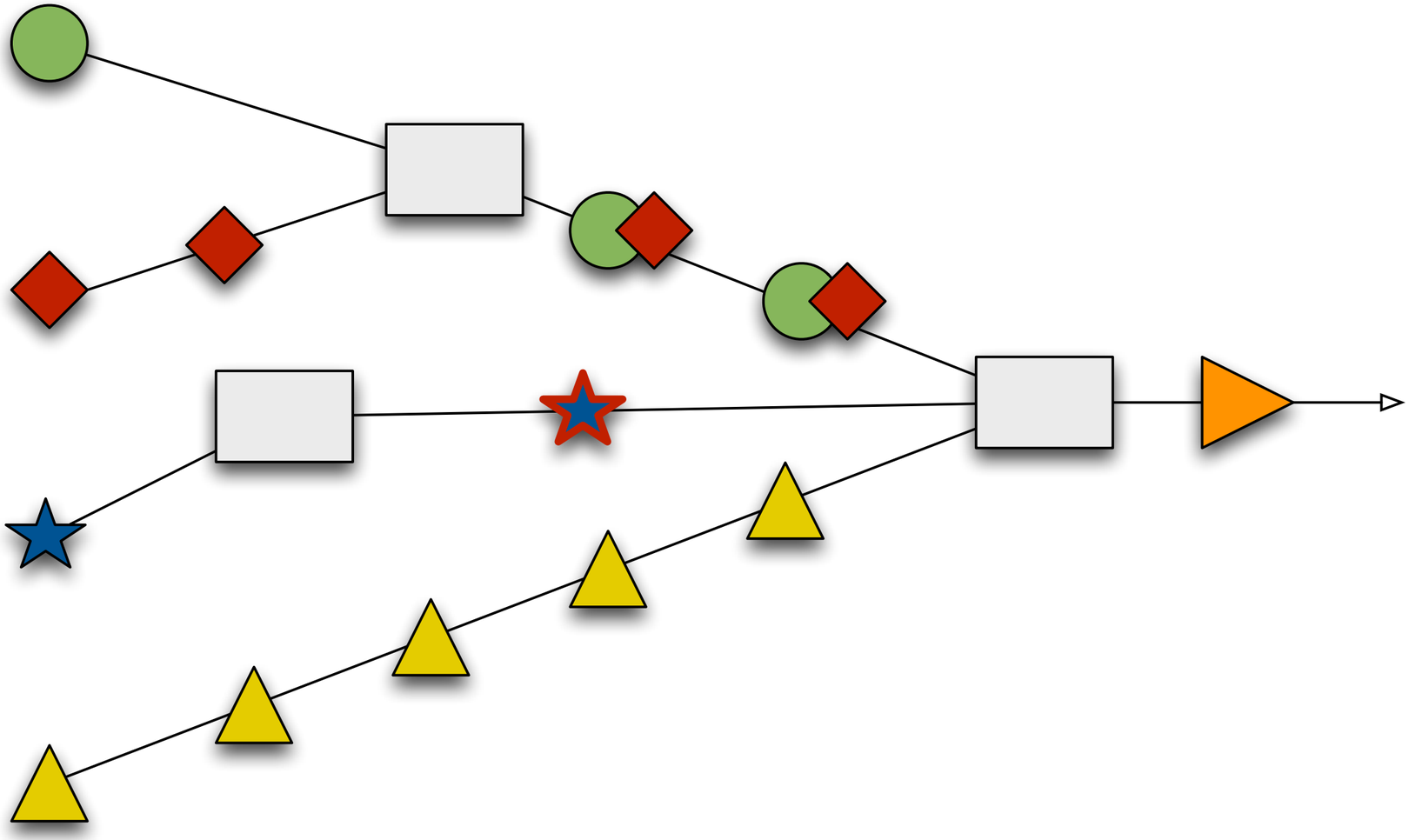
Asynchronous Observer

```
getVideos().subscribe(
    { video ->
        println("Video: " + video.videoId)
    }, { exception ->
        println("Error")
        e.printStackTrace()
    }, {
        println("Completed")
    }
)
```

Asynchronous Observer

```
getVideos().subscribe(
    { video ->
        println("Video: " + video.videoId)
    }, { exception ->
        println("Error")
        e.printStackTrace()
    }
)
```

COMPOSABLE FUNCTIONS



COMPOSABLE FUNCTIONS

TRANSFORM: MAP, FLATMAP, REDUCE, SCAN …

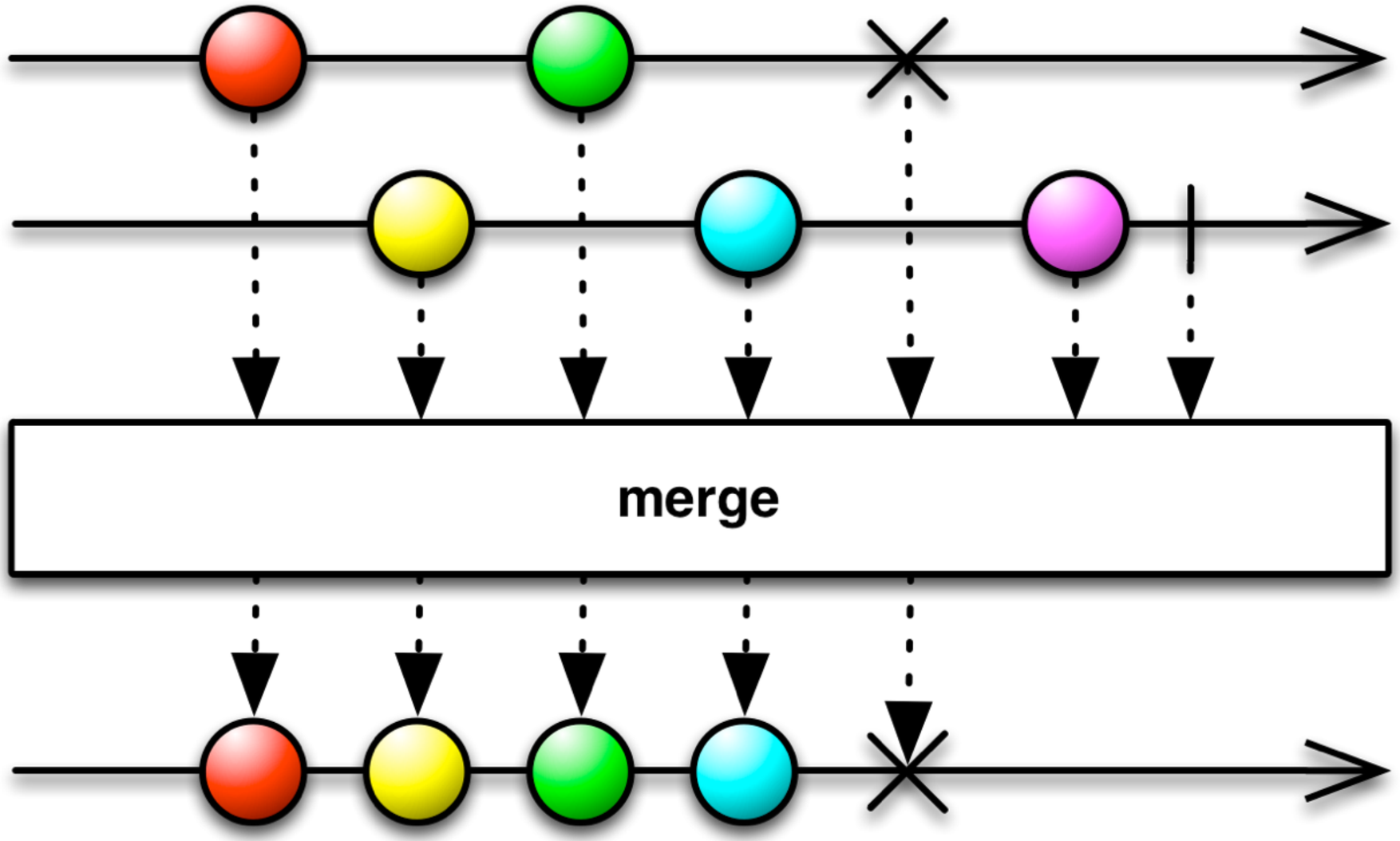
FILTER: TAKE, SKIP, SAMPLE, TAKEWHILE, FILTER …

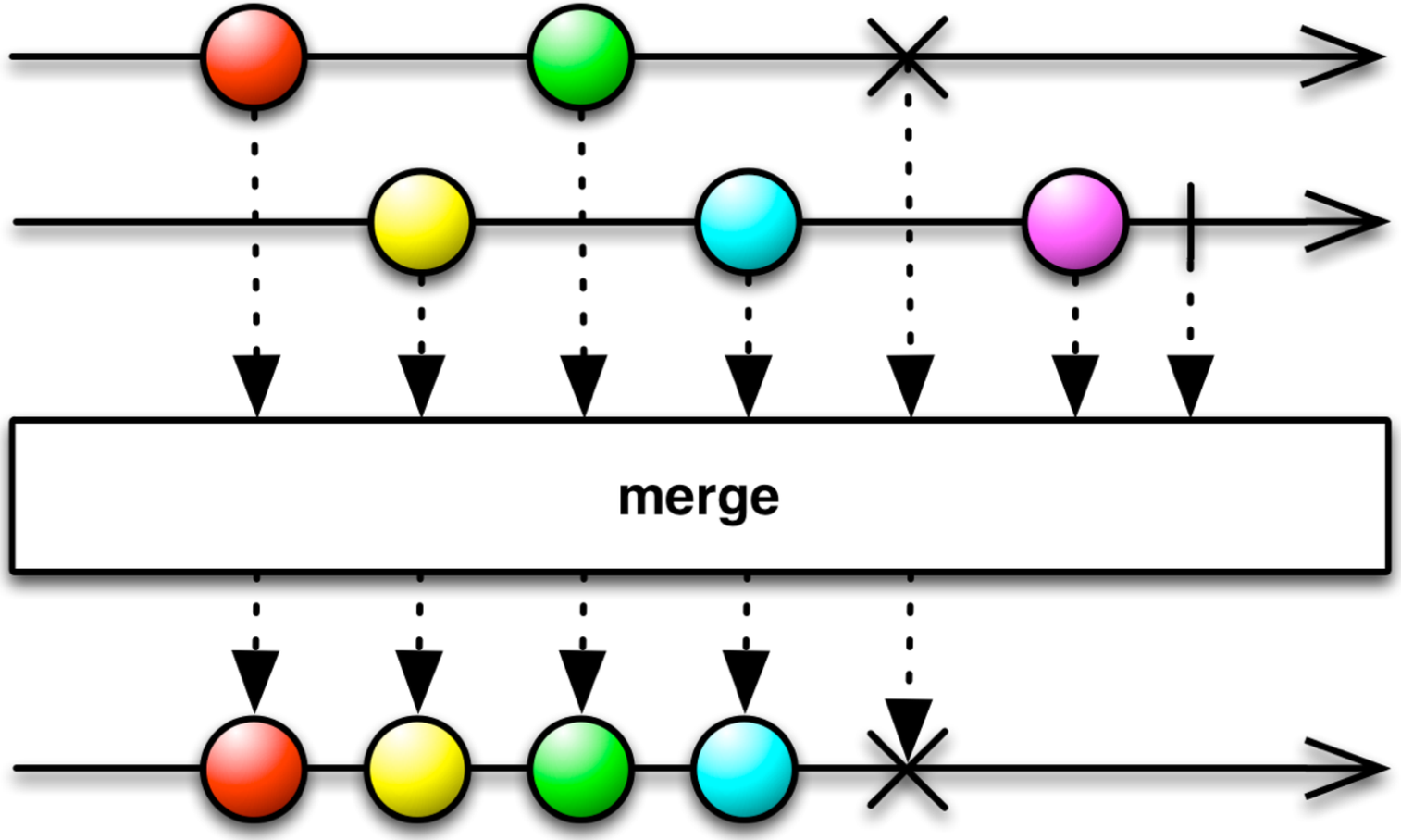
COMBINE: CONCAT, MERGE, ZIP, COMBINELATEST,
MULTICAST, PUBLISH, CACHE, REFCOUNT …

CONCURRENCY: OBSERVEON, SUBSCRIBEON

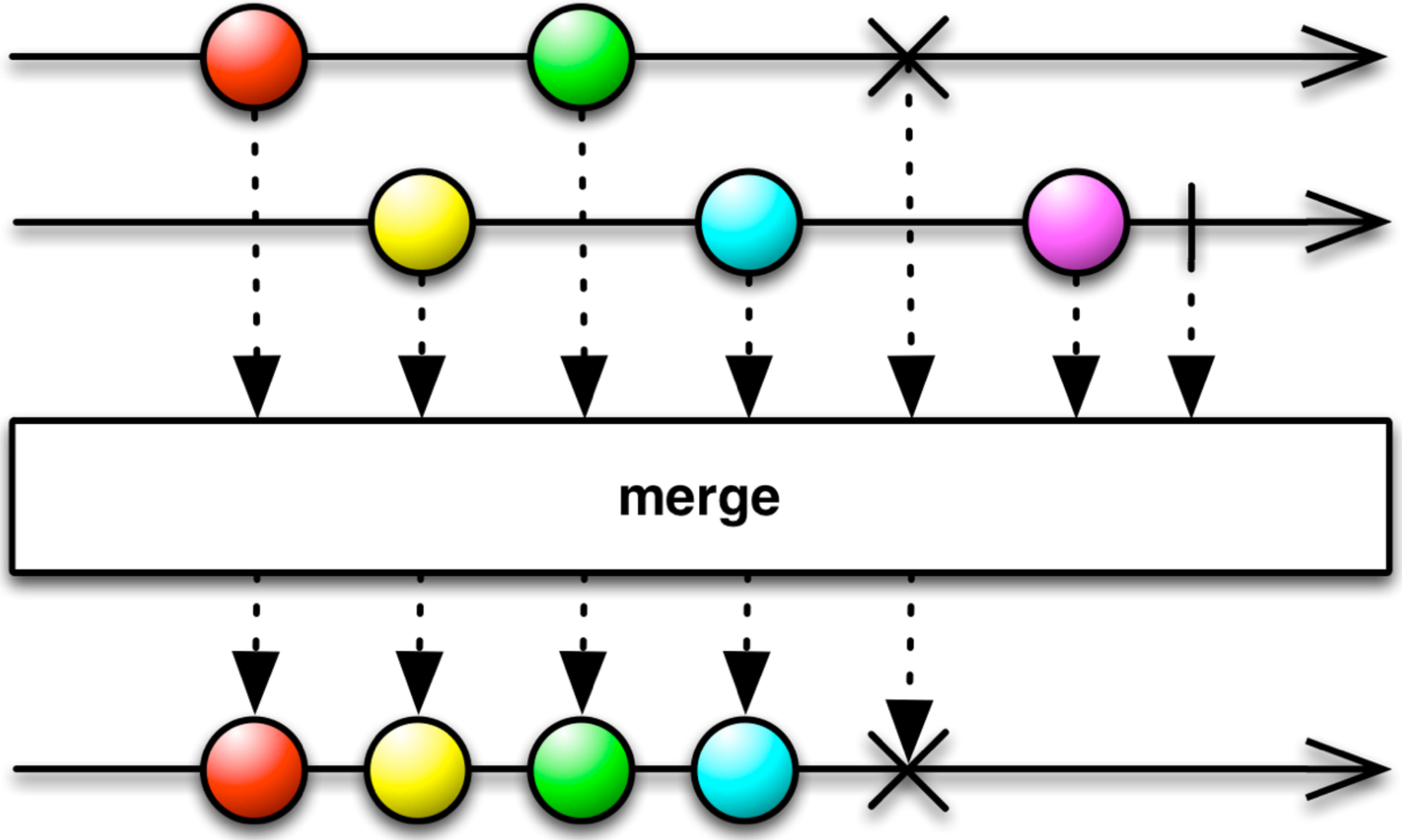
ERROR HANDLING: ONERRORRETURN, ONERRORRESUME …

Combining via Merge



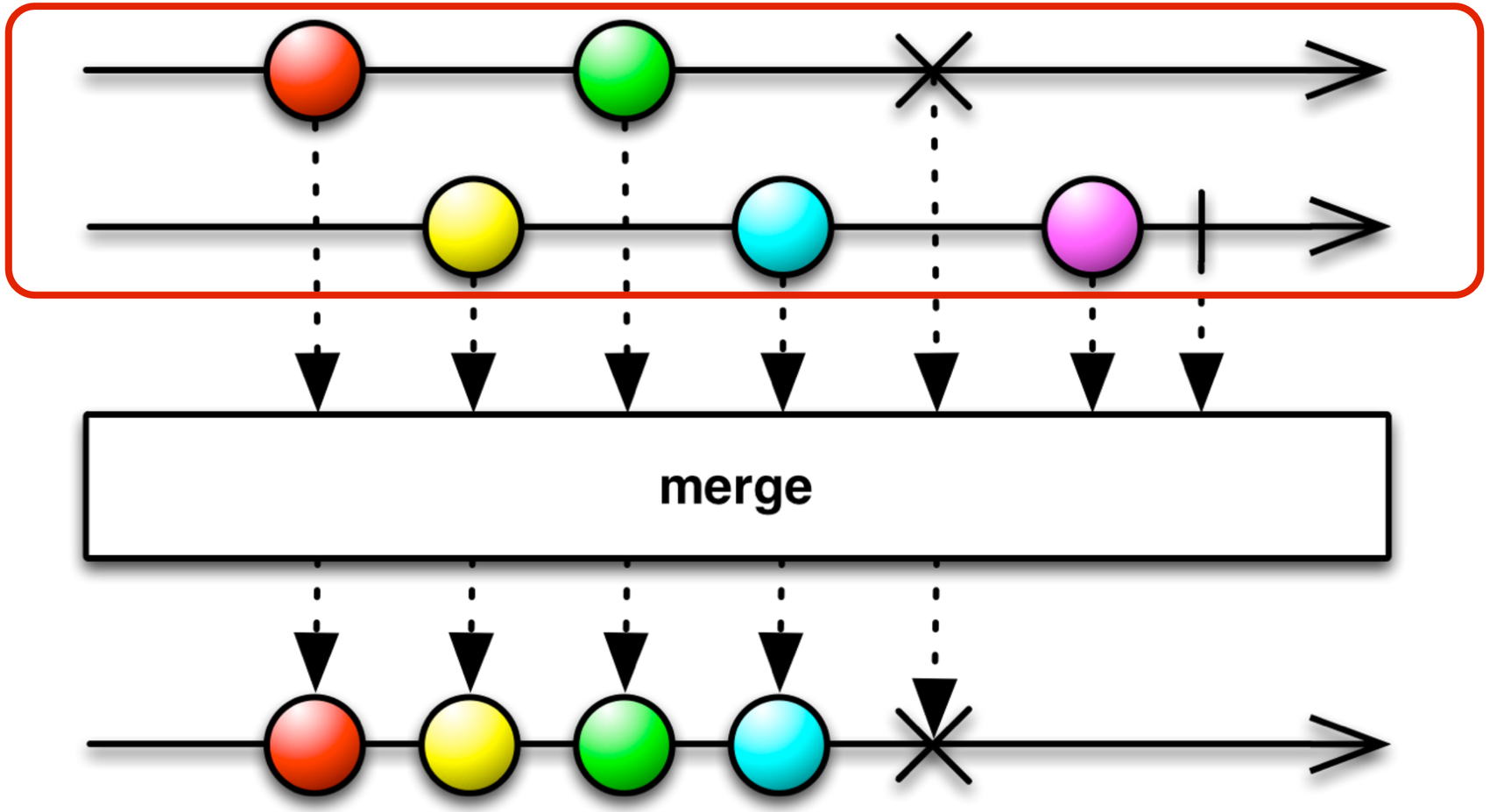



```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();

Observable.merge(a, b)
    .subscribe(
        { element -> println("data: " + element)})
```

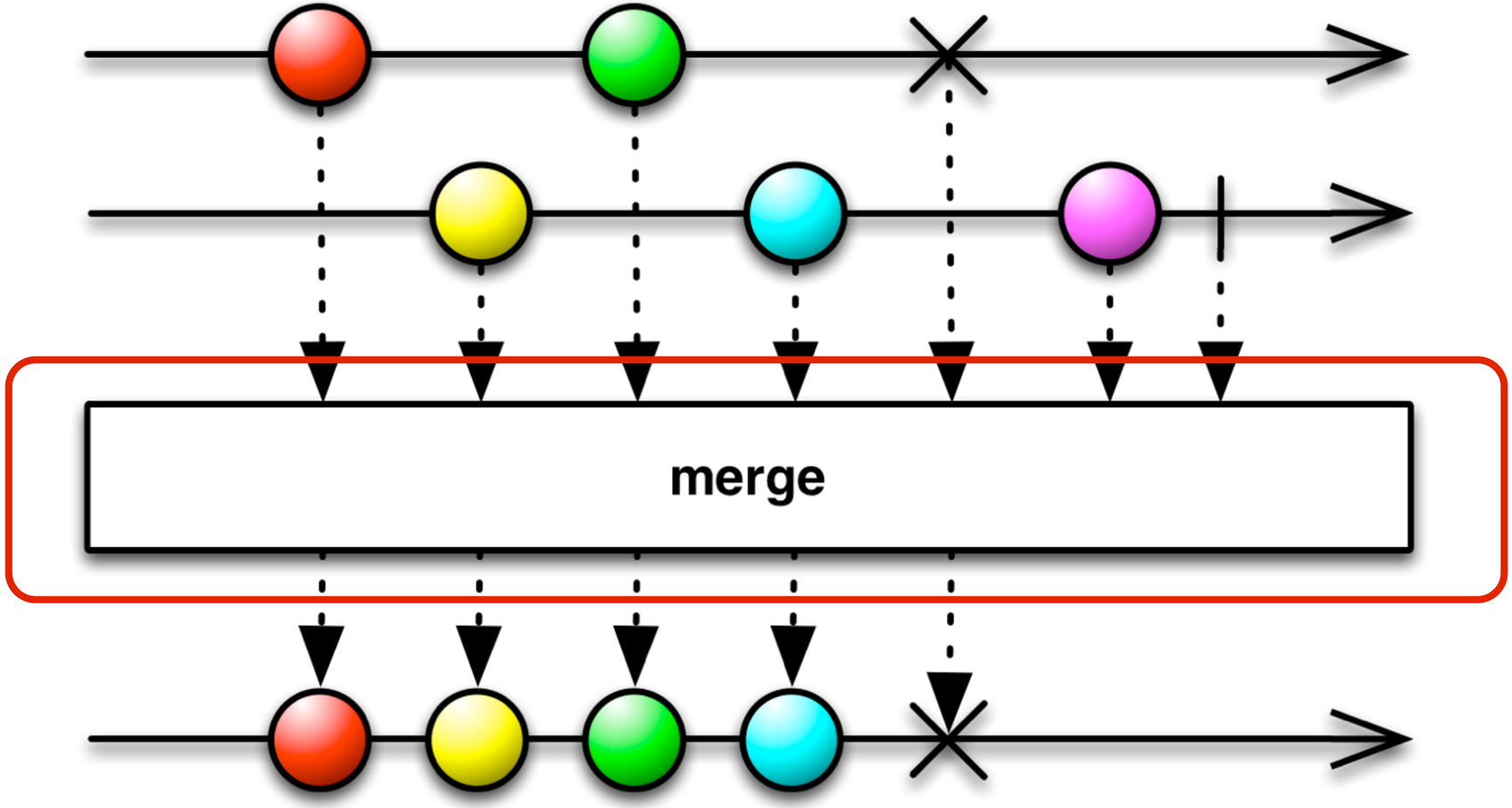



```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();
```

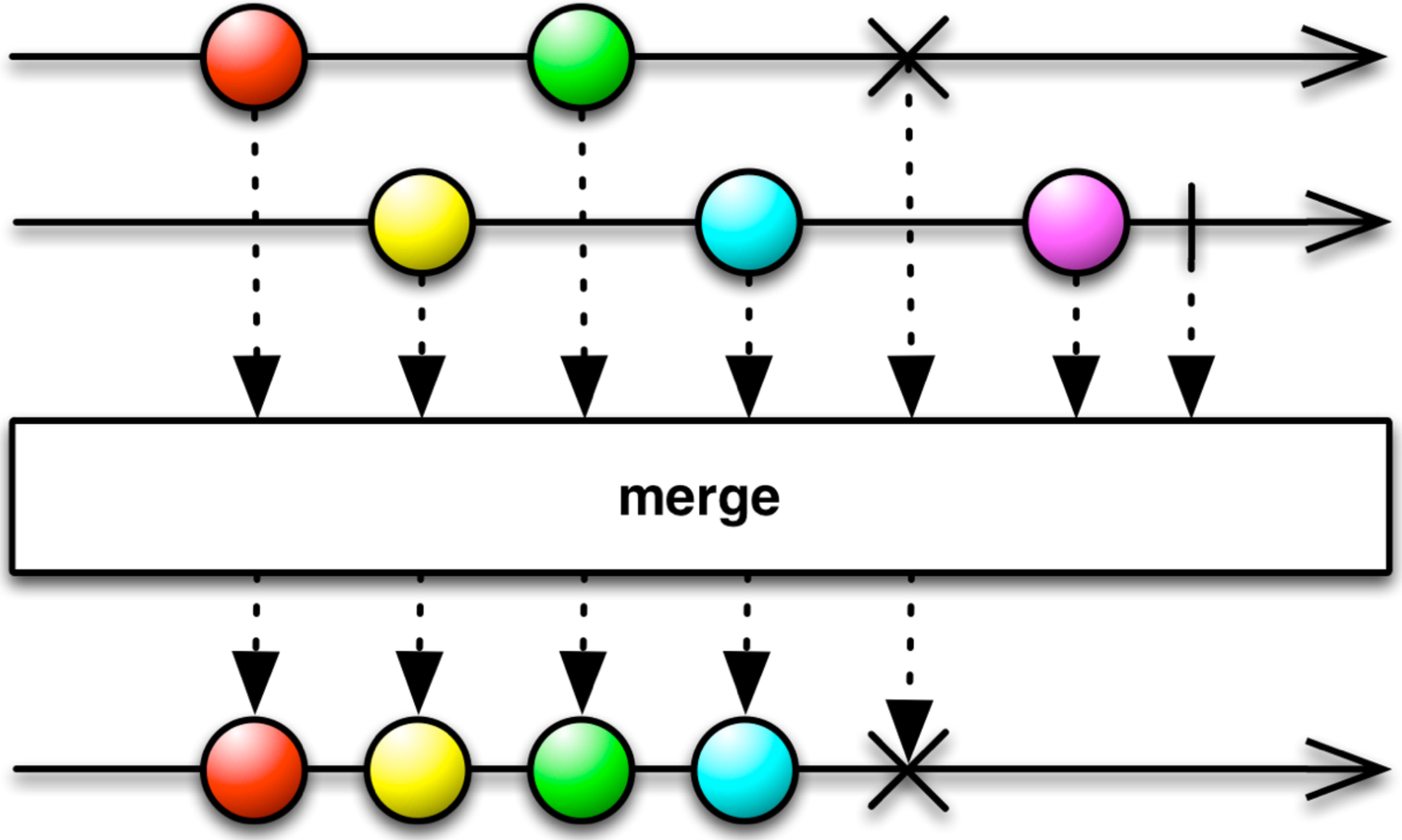
```
Observable.merge(a, b)
    .subscribe(
        { element -> println("data: " + element)})
```



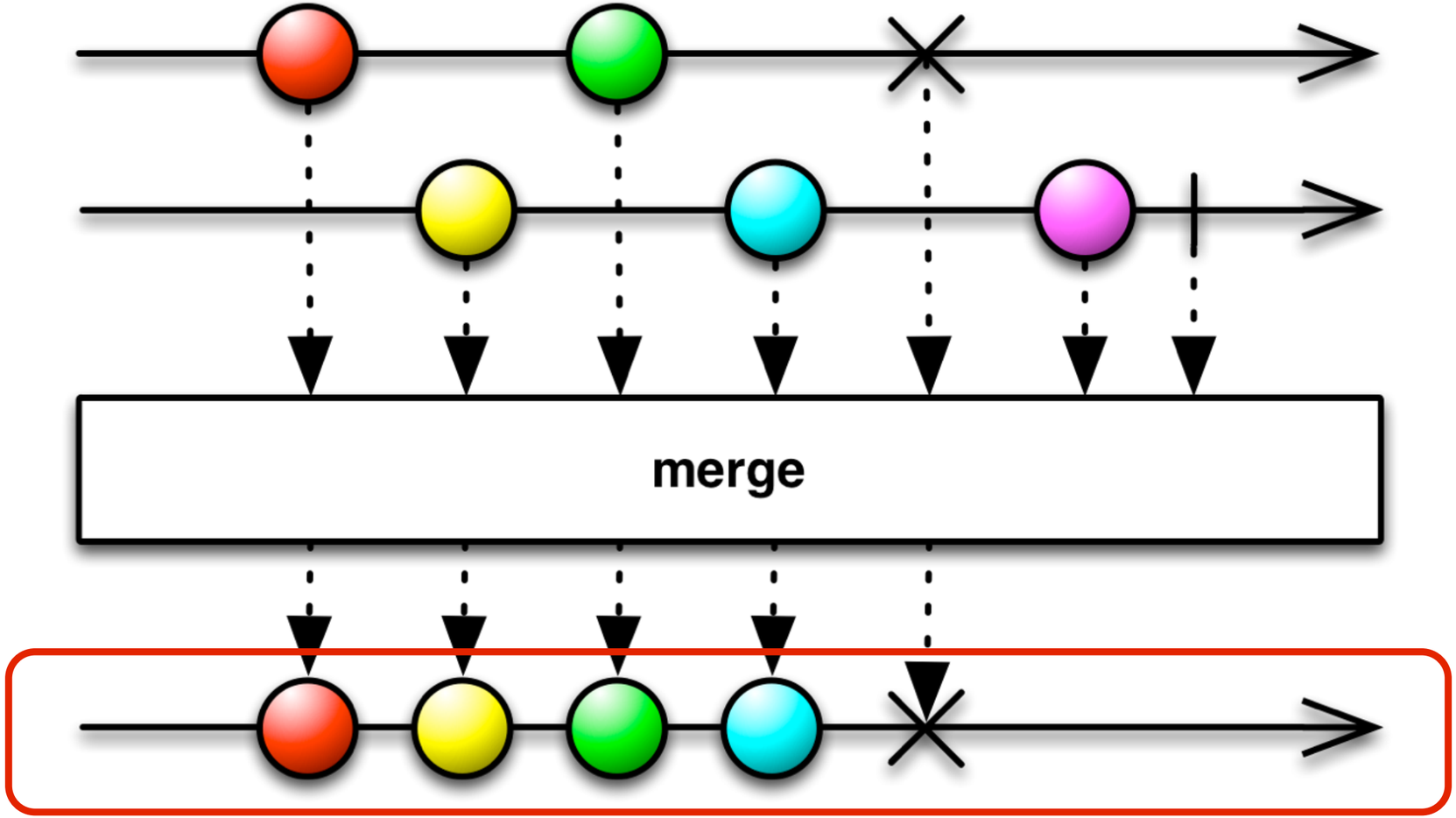

```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();

Observable.merge(a, b)
    .subscribe(
        { element -> println("data: " + element)})
```



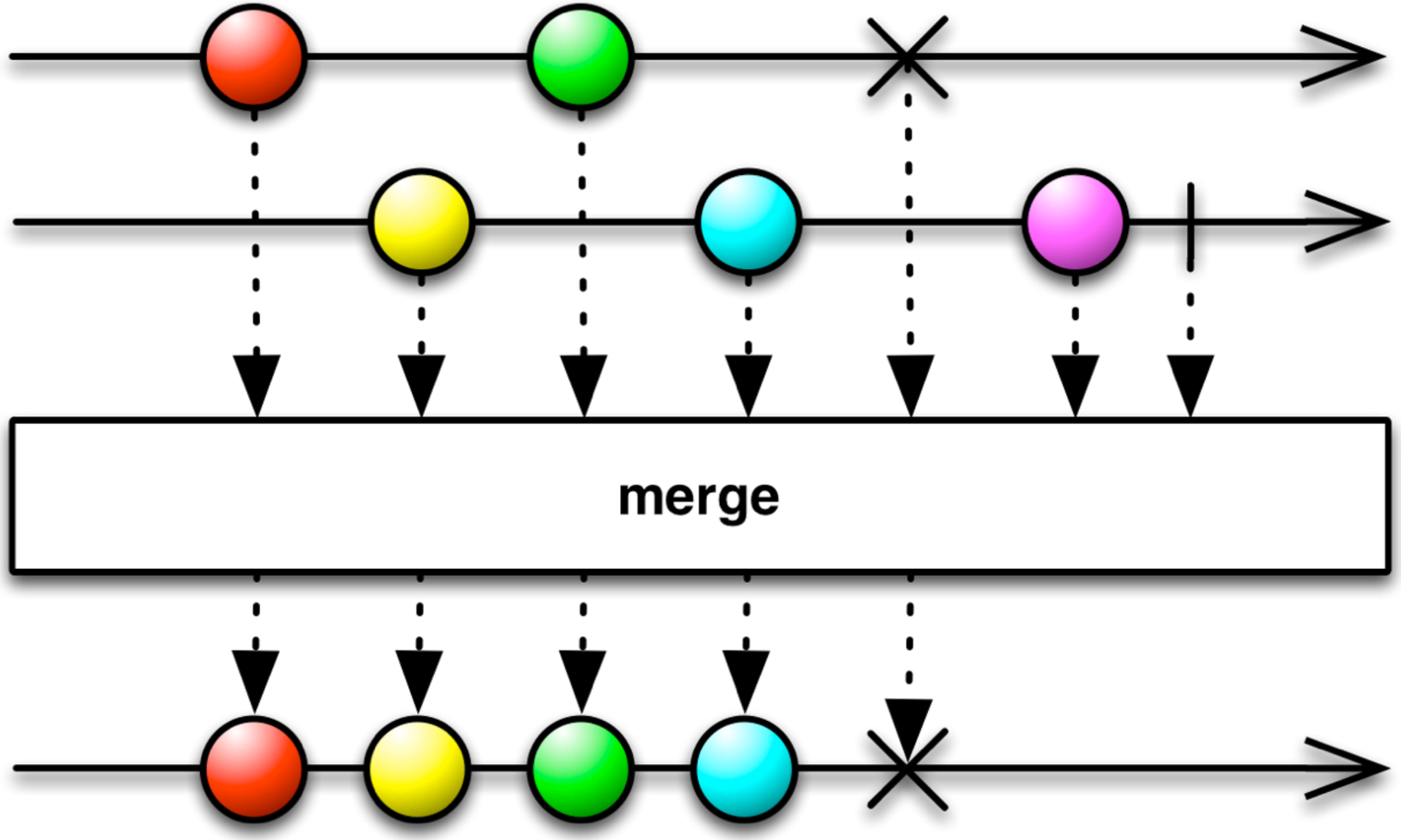

```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();

Observable.merge(a, b)
    .subscribe(
        { element -> println("data: " + element)})
```




```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();

Observable.merge(a, b)
        .subscribe(
                { element -> println("data: " + element)})
```

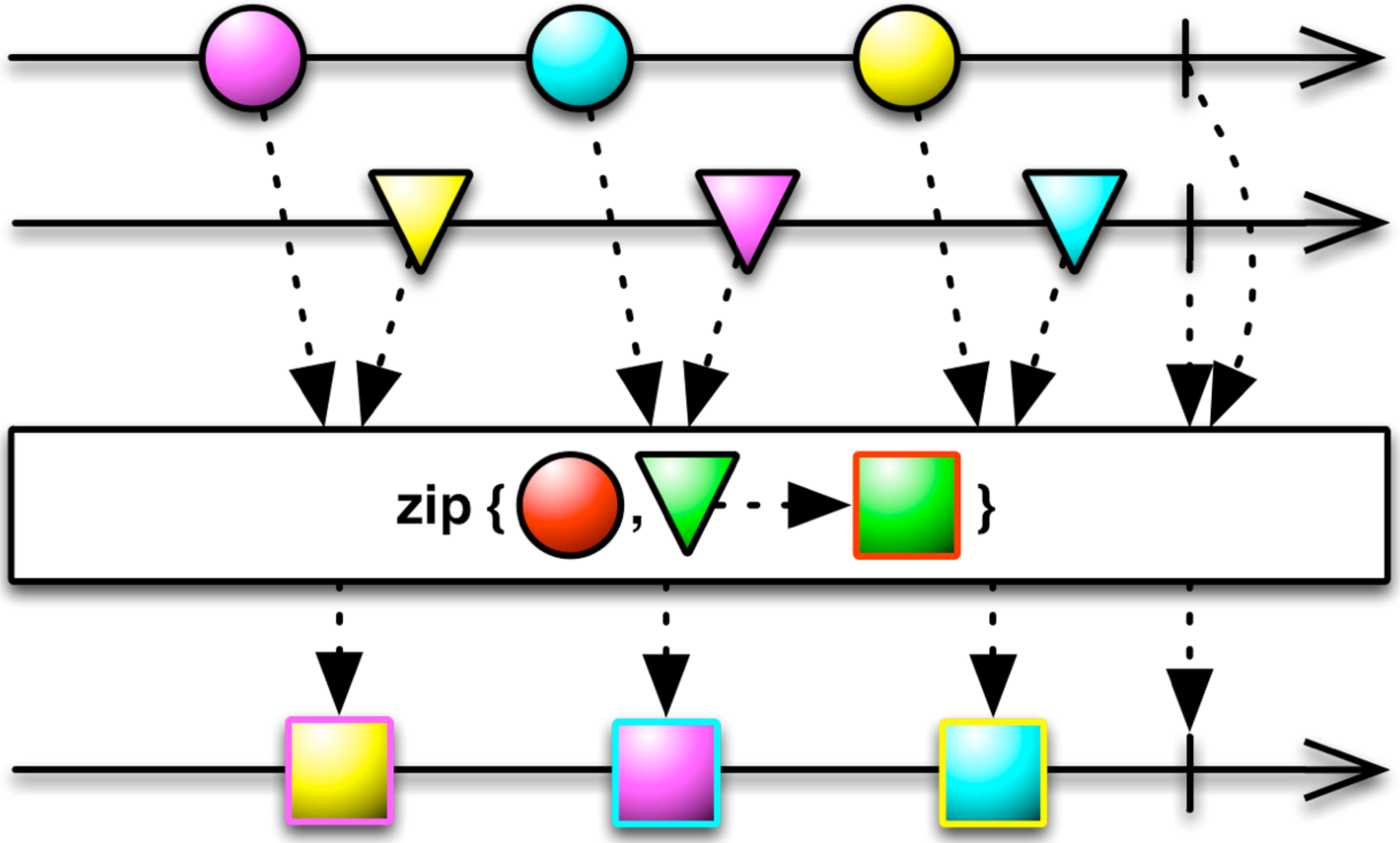


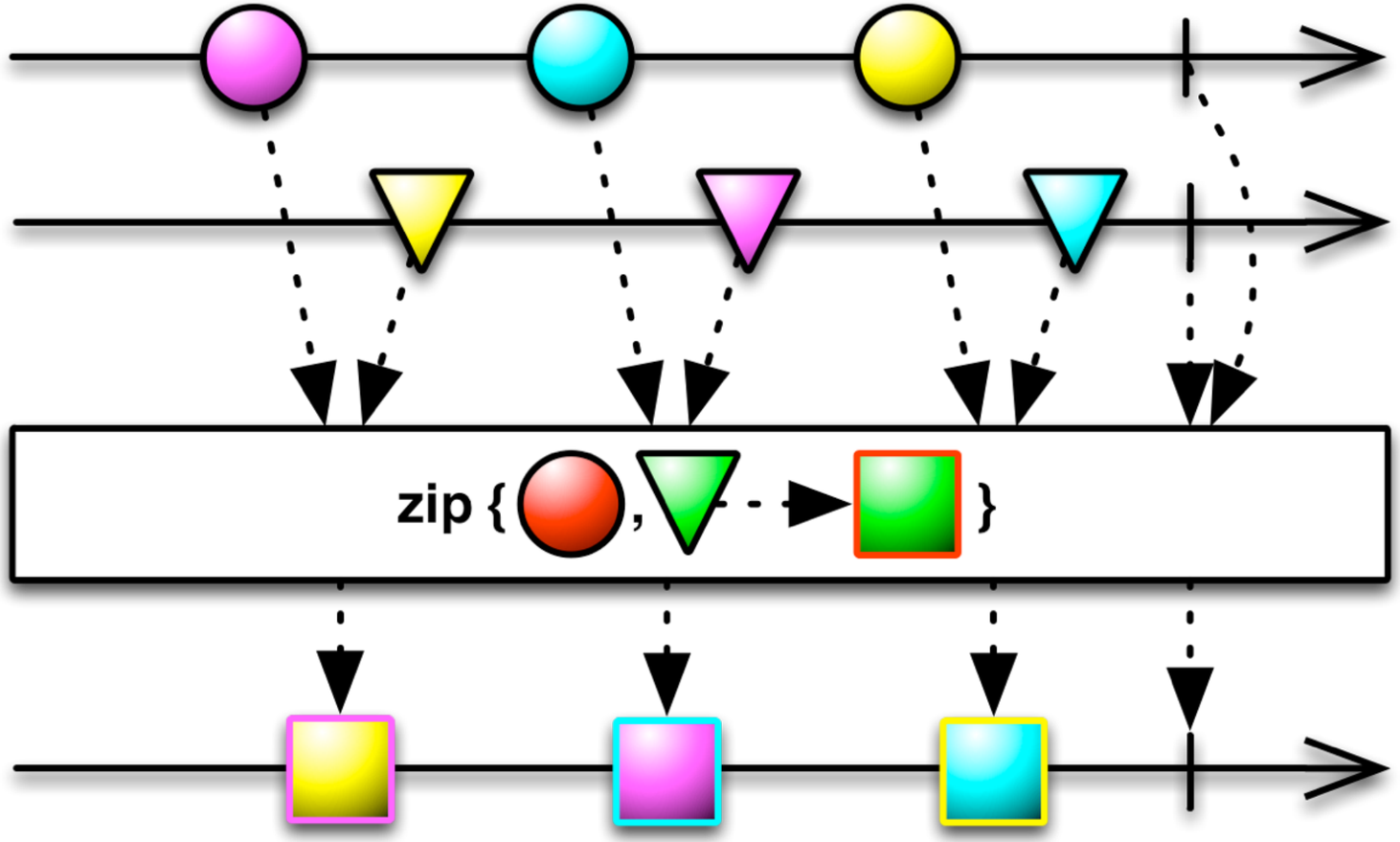

```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();

Observable.merge(a, b)
    .subscribe(
        { element -> println("data: " + element)})
```



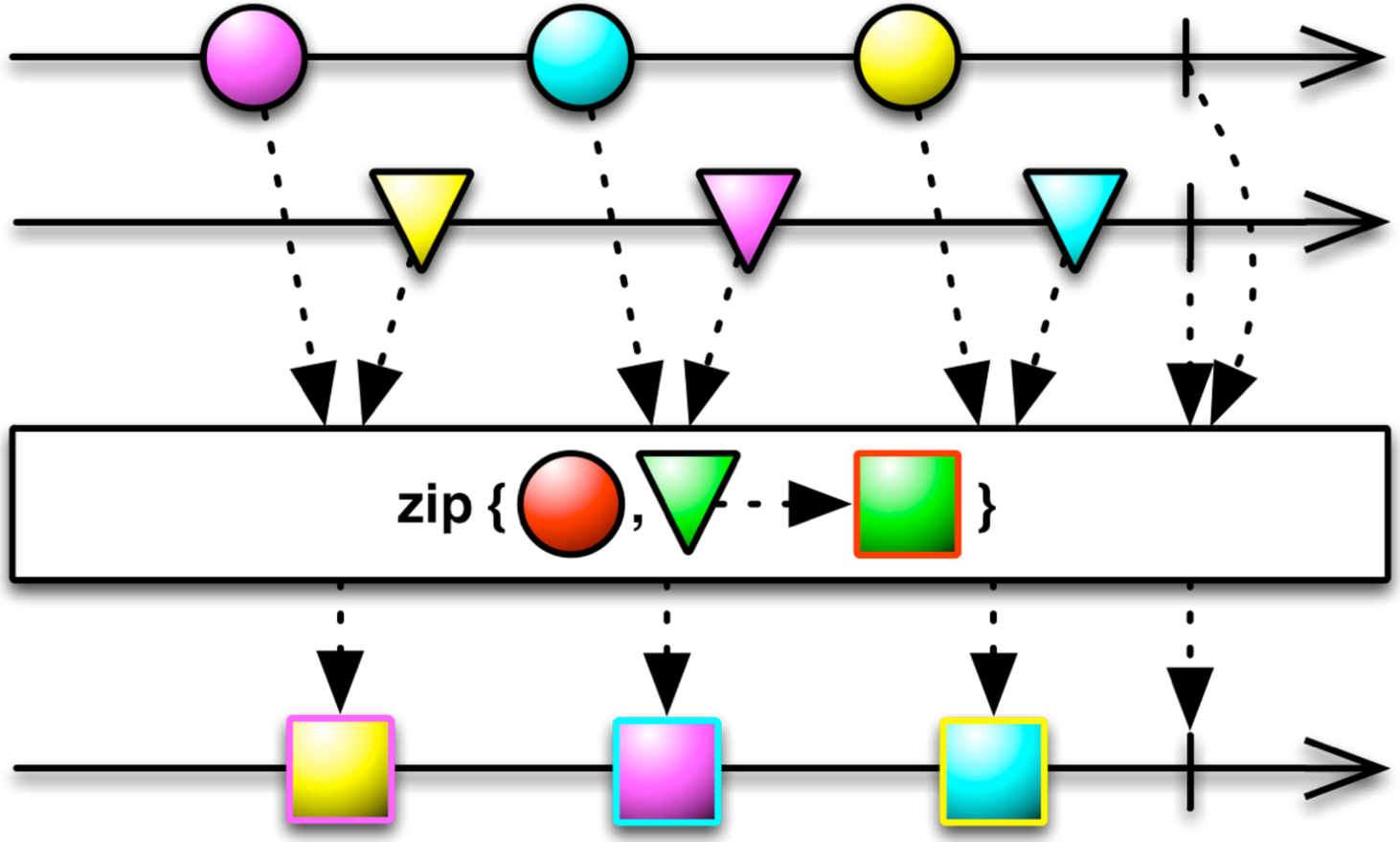

```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();

Observable.merge(a, b)
    .subscribe(
      { element -> println("data: " + element)})
```

Combining via Zip



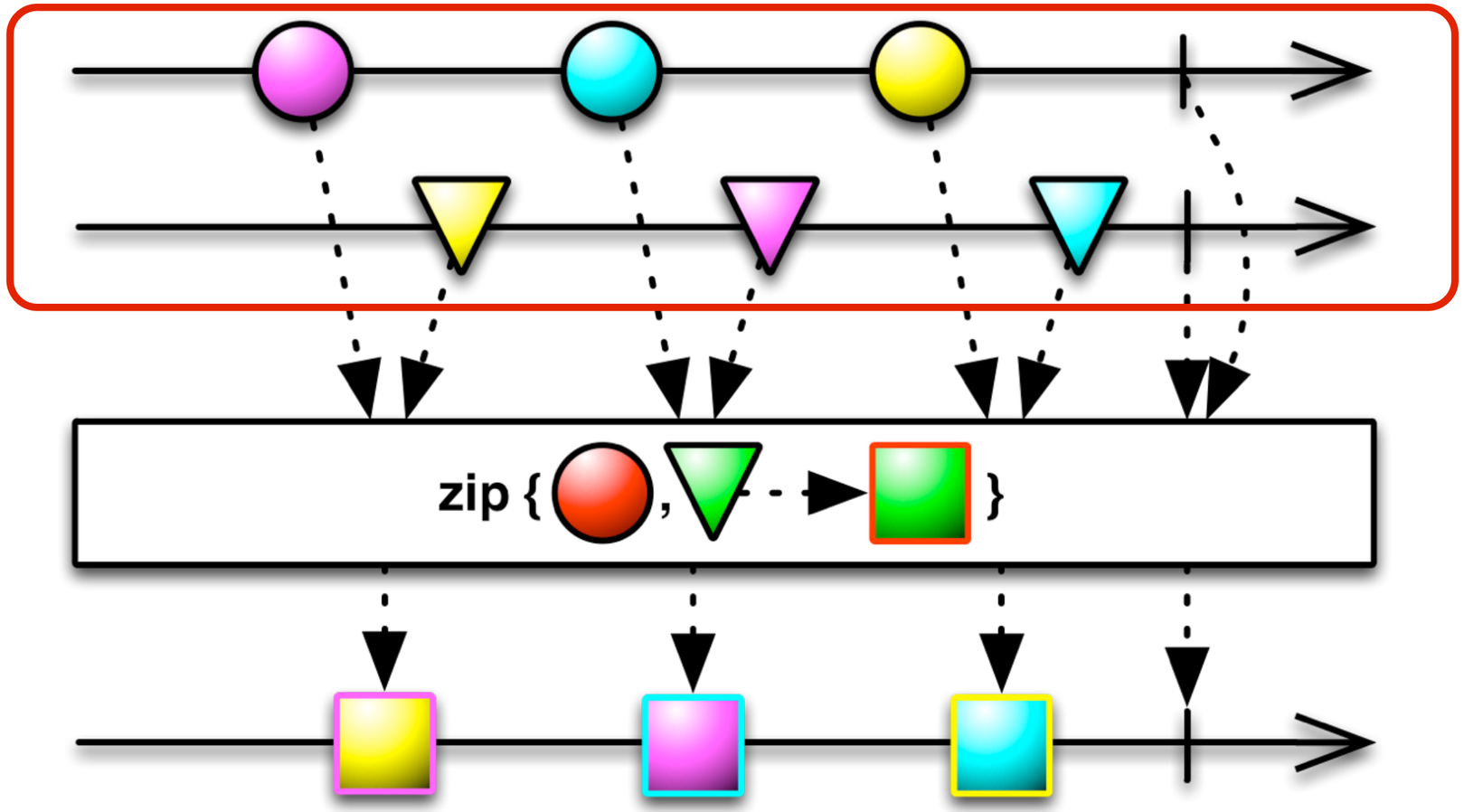



```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])})
```

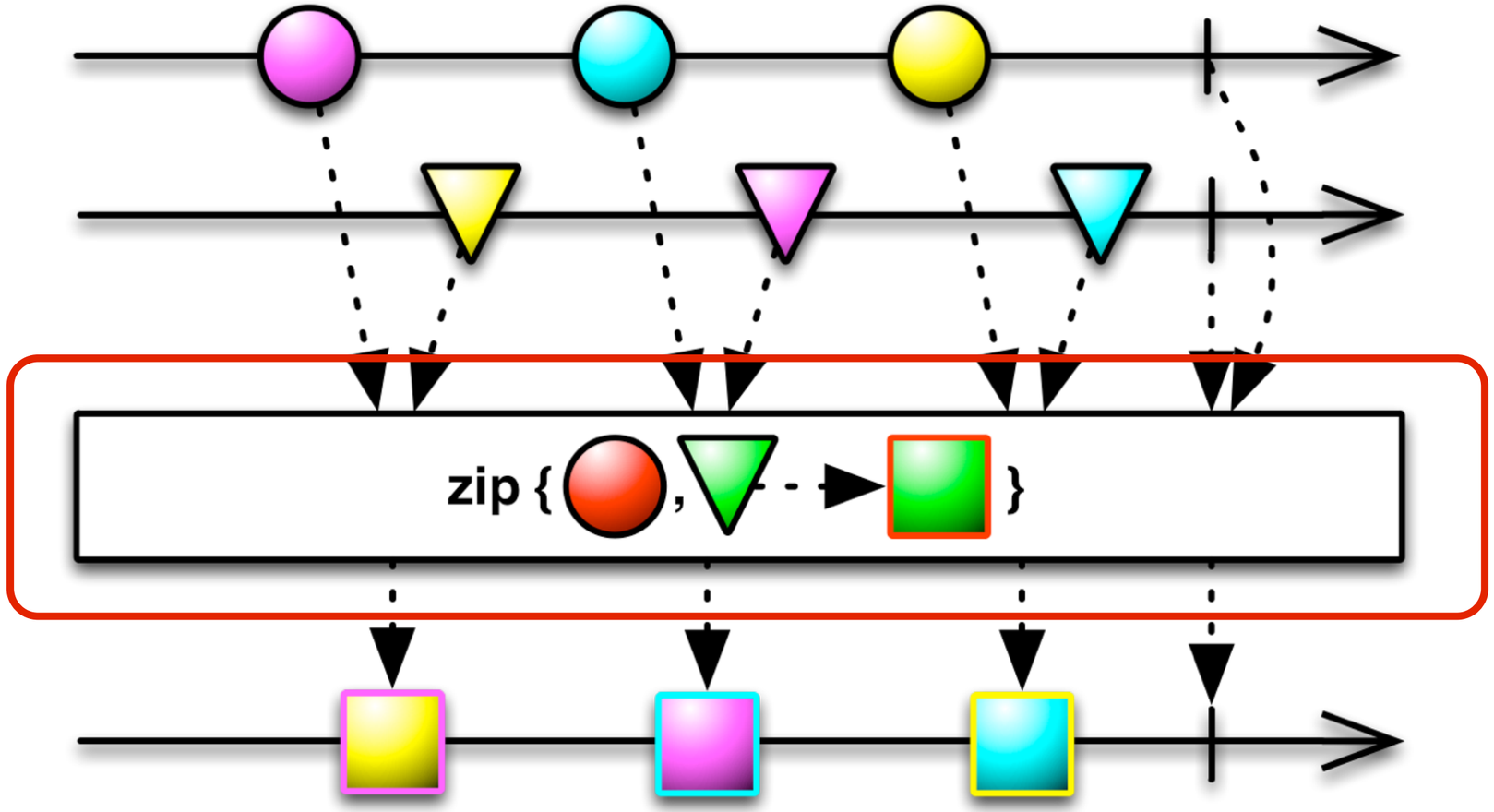



```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();
```

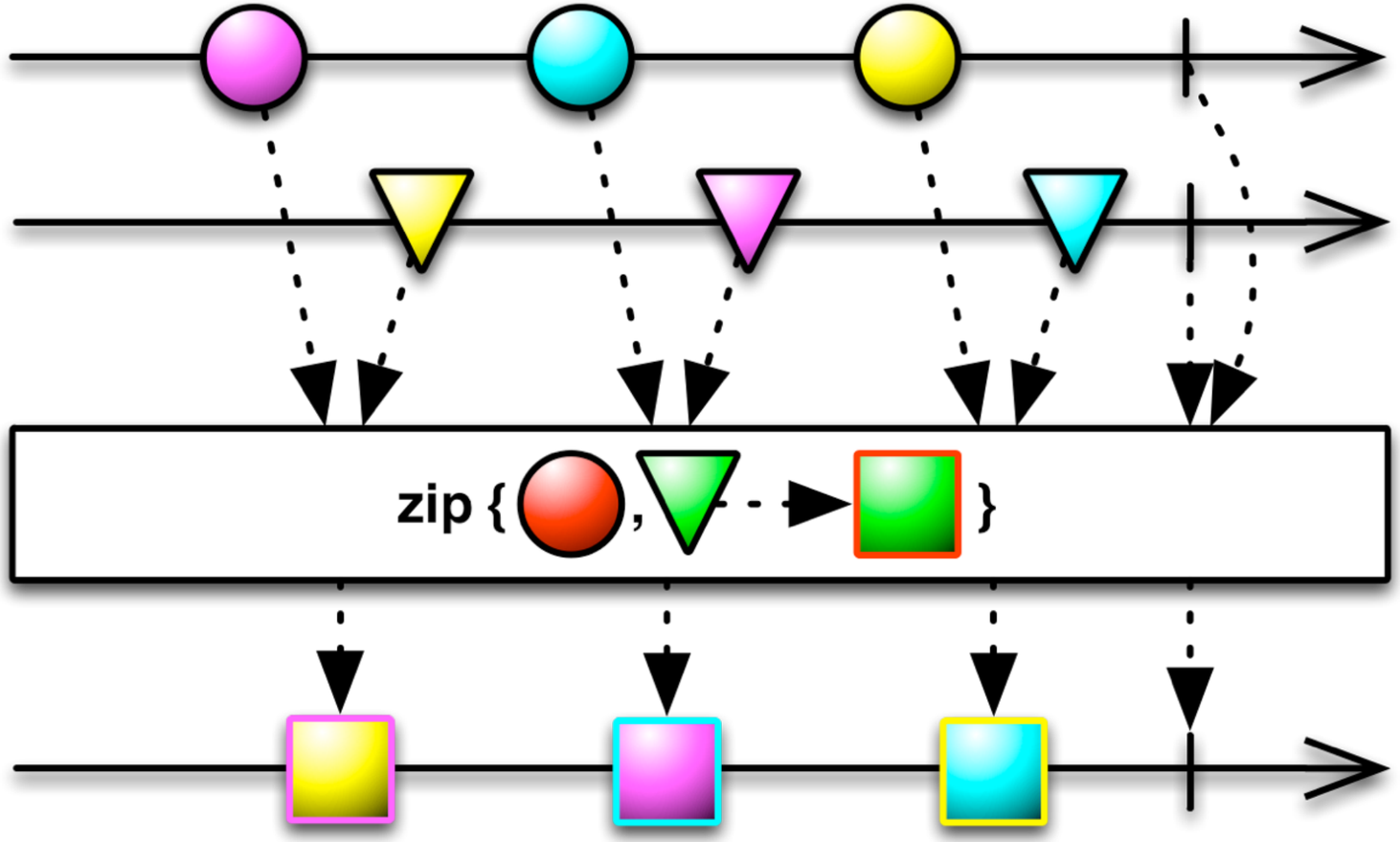
```
Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])})
```



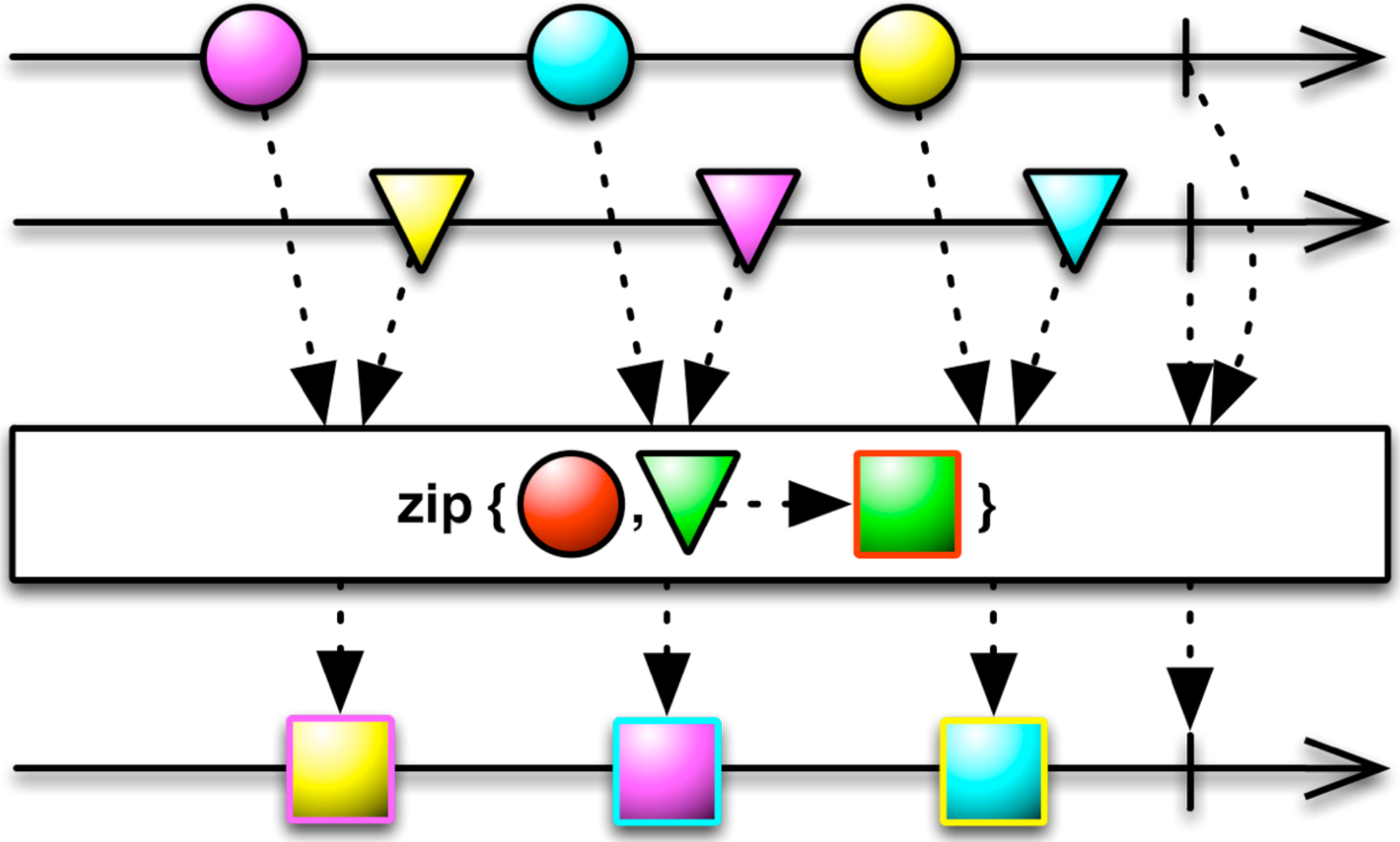

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])})
```



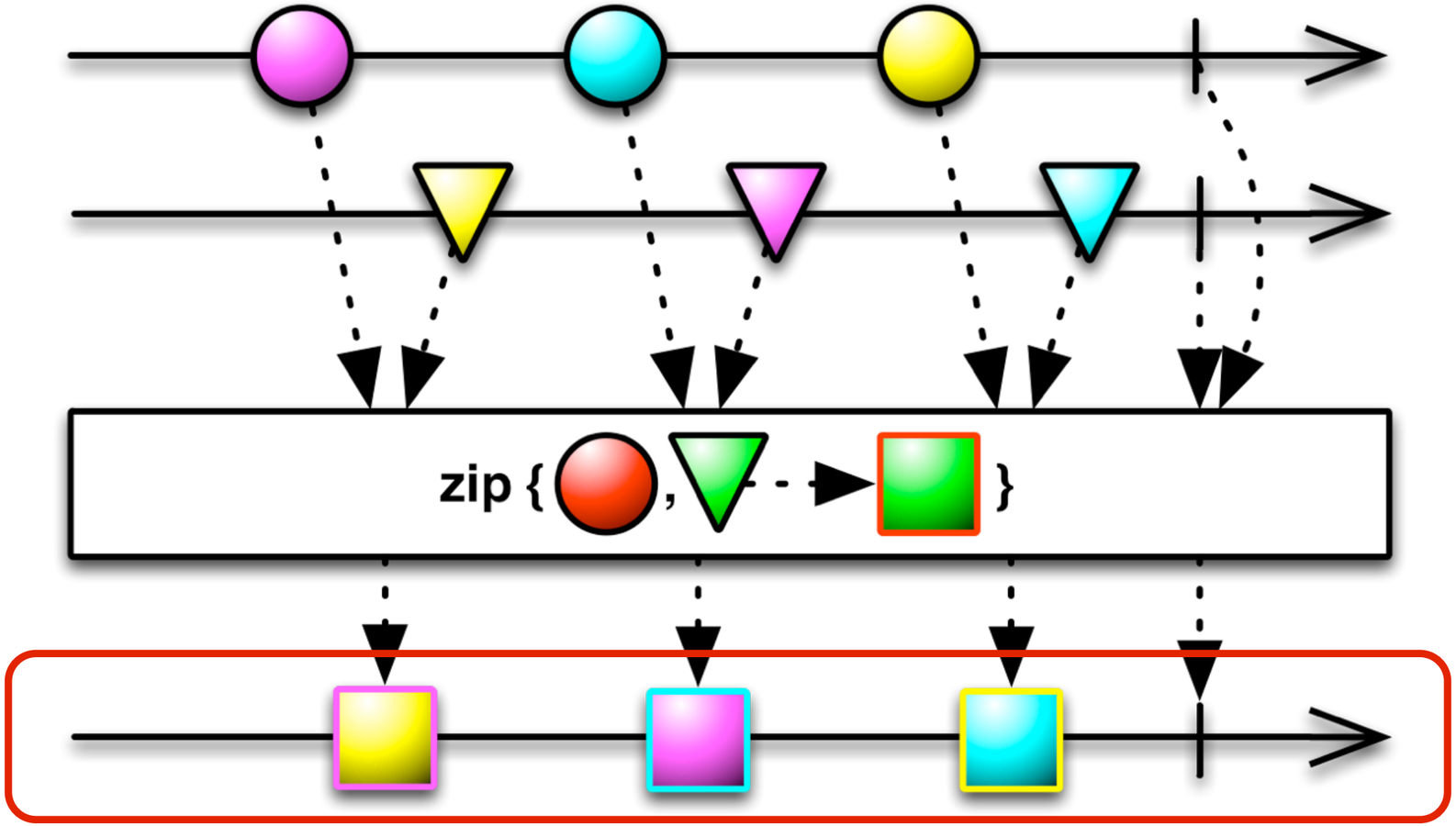

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])})
```



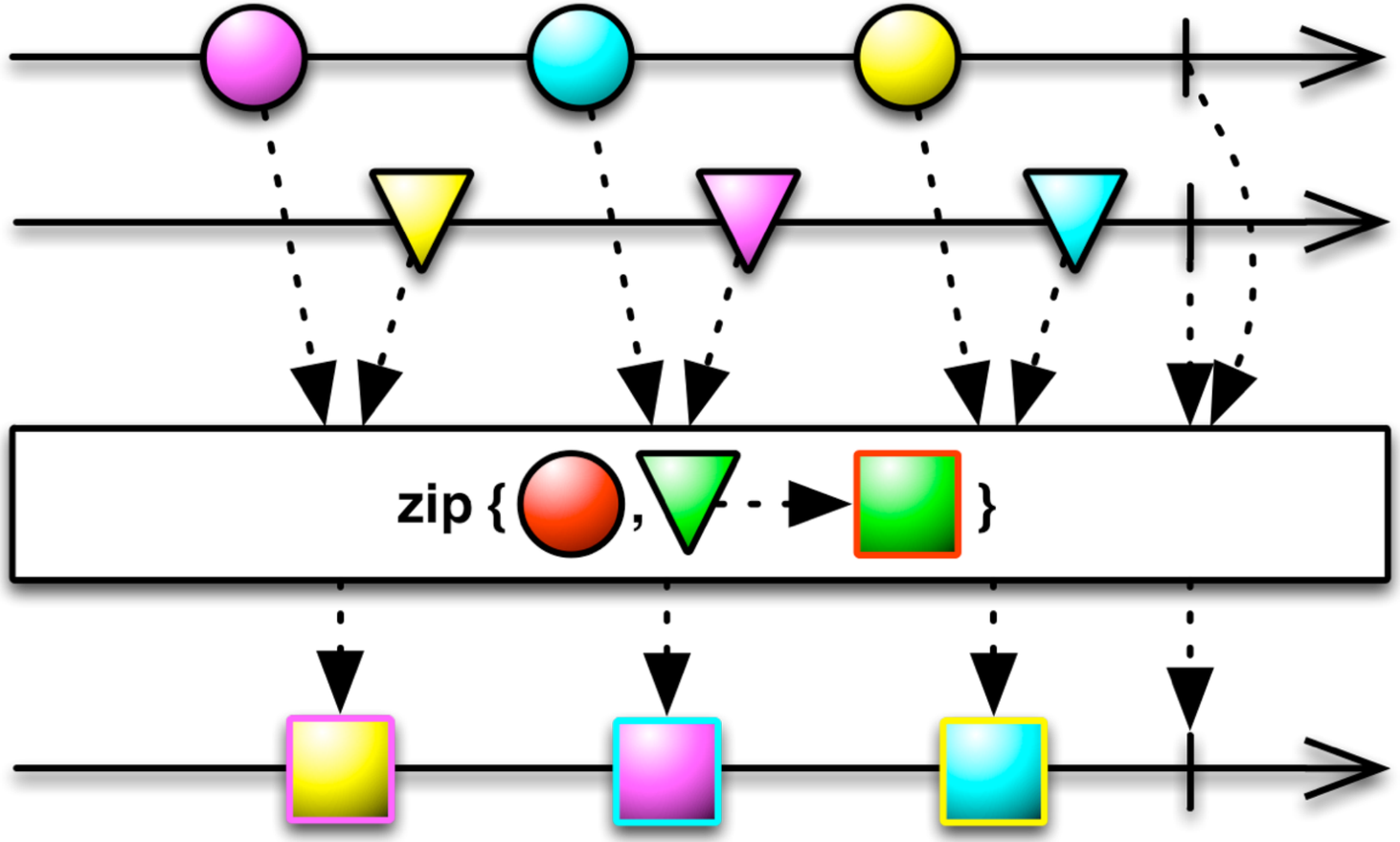

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

Observable.zip(a, b, {x, y -> [x, y]})
      .subscribe(
          { pair -> println("a: " + pair[0]
                          + " b: " + pair[1])})
```




```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])})
```




```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])})
```




```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])})
```

Error Handling

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])},
        { exception -> println("error occurred: "
                               + exception.getMessage())},
        { println("completed") })
```

Error Handling

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])},
        { exception -> println("error occurred: "
                               + exception.getMessage())},
        { println("completed") })
```

onNext(T)

onError(Exception)

onCompleted()

Error Handling

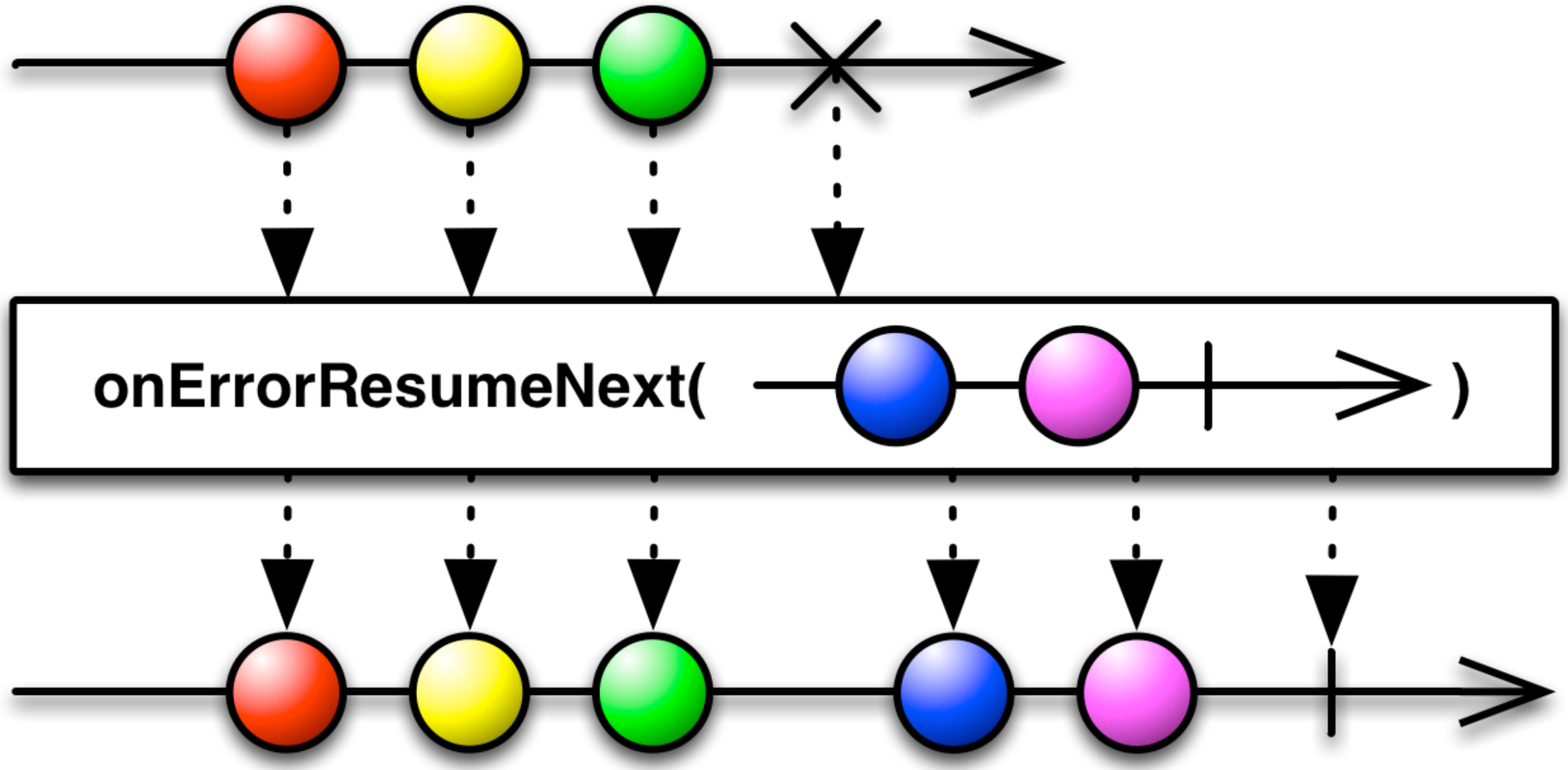
```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])},
        { exception -> println("error occurred: "
                               + exception.getMessage())},
        { println("completed") })
```

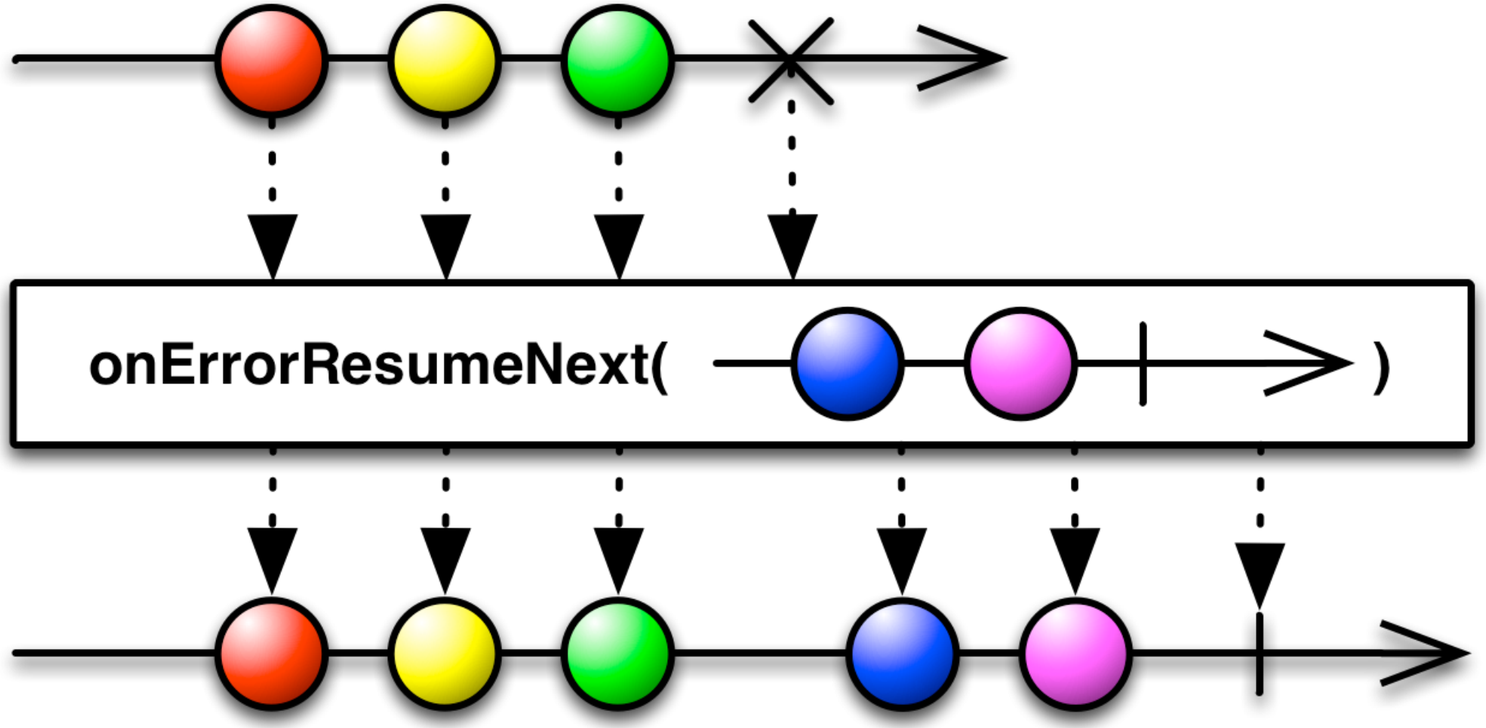
onNext(T)

onError(Exception)

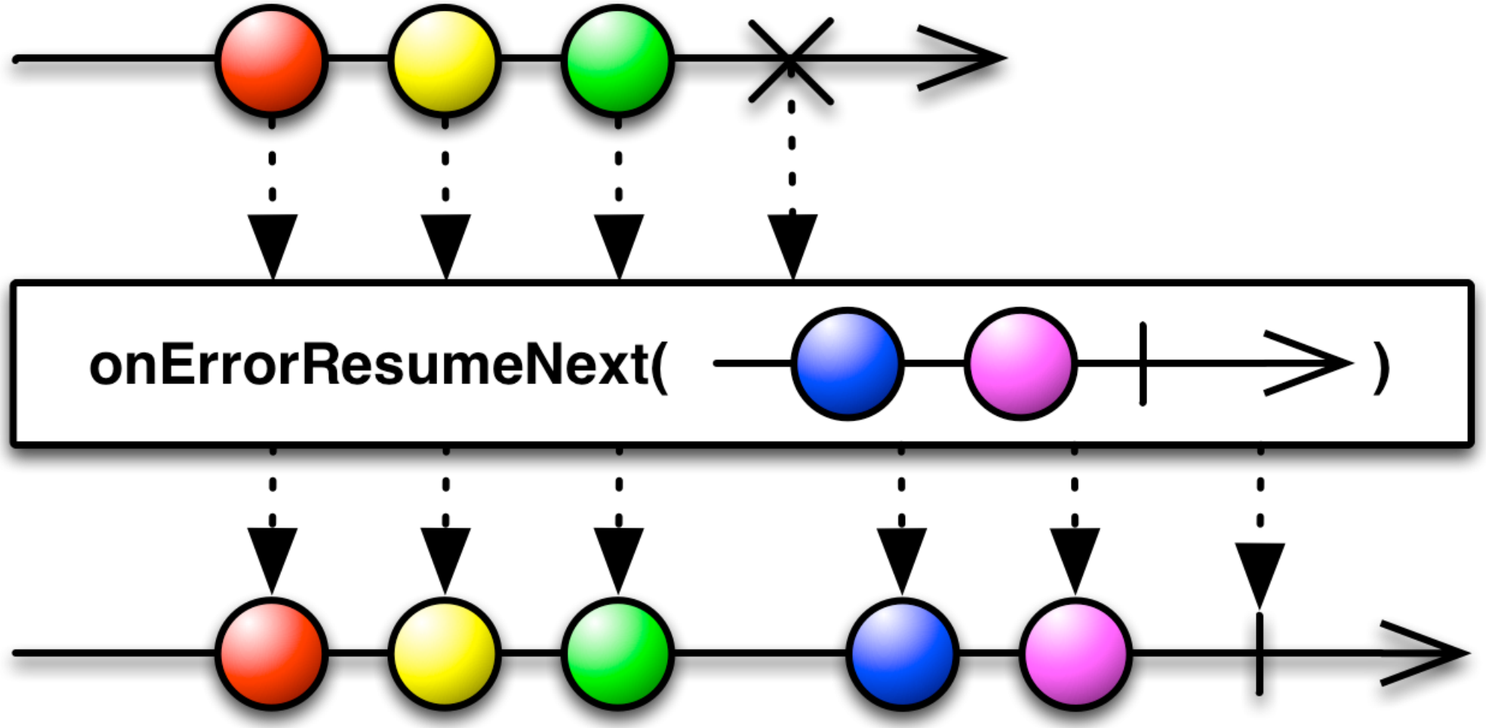
onCompleted()

Error Handling

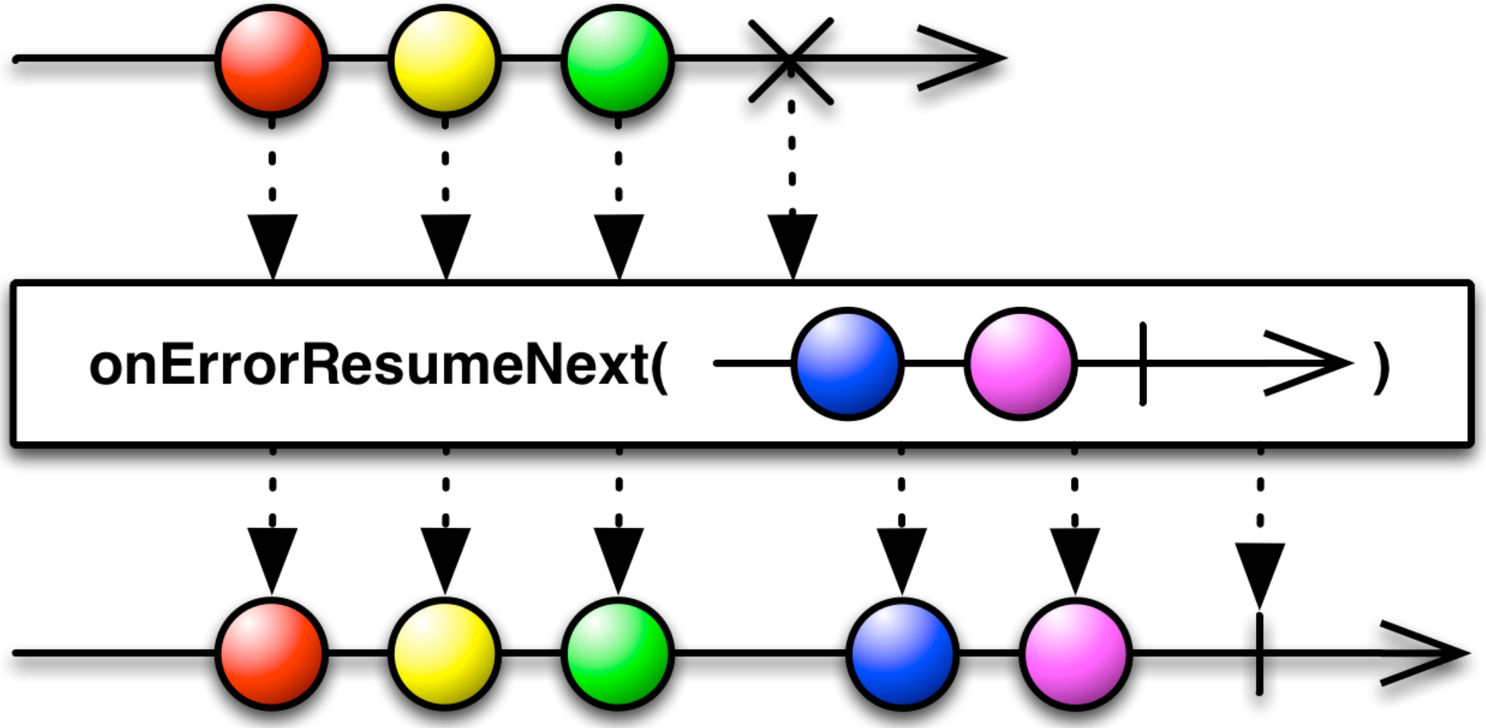




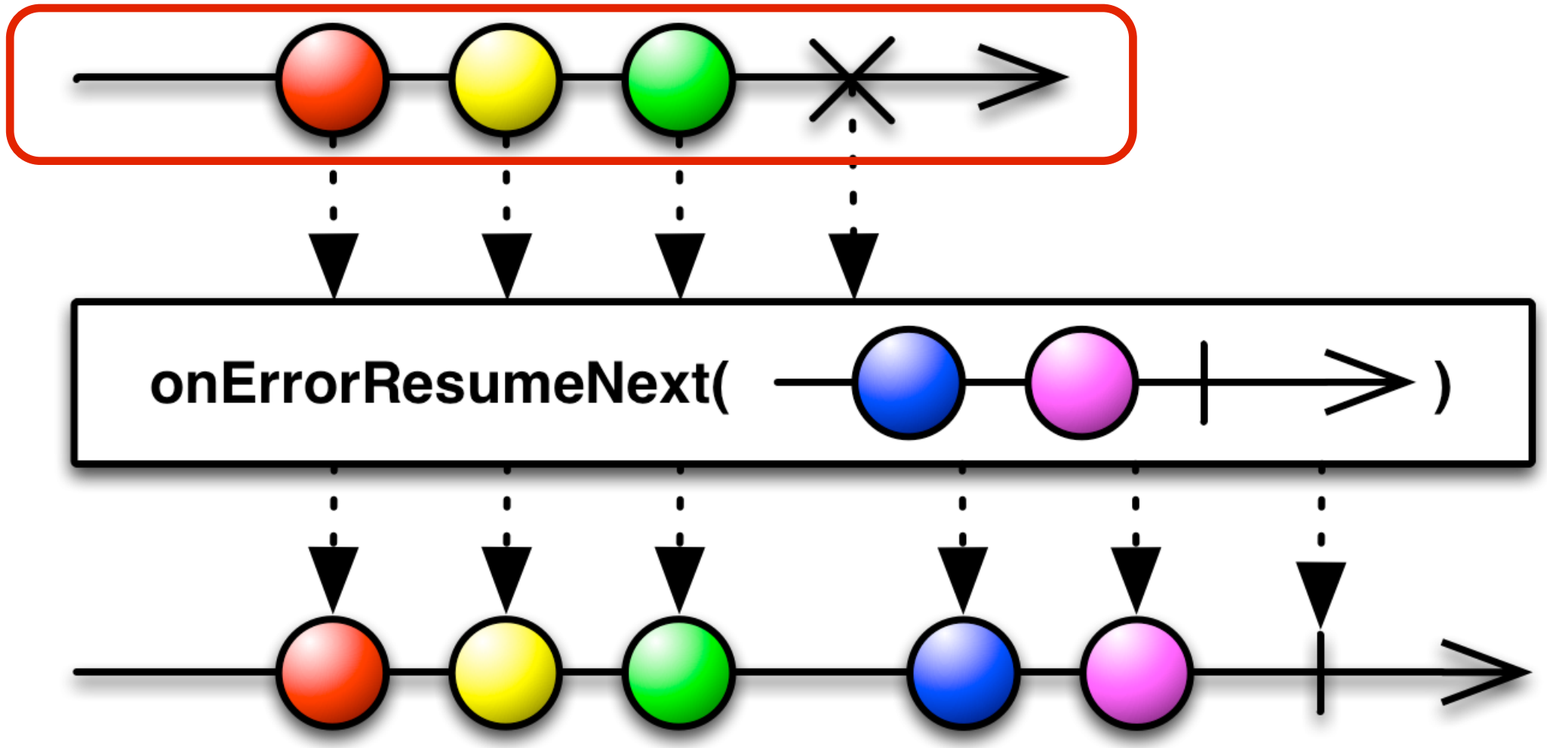

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])},
        { exception -> println("error occurred: "
                               + exception.getMessage())})
```



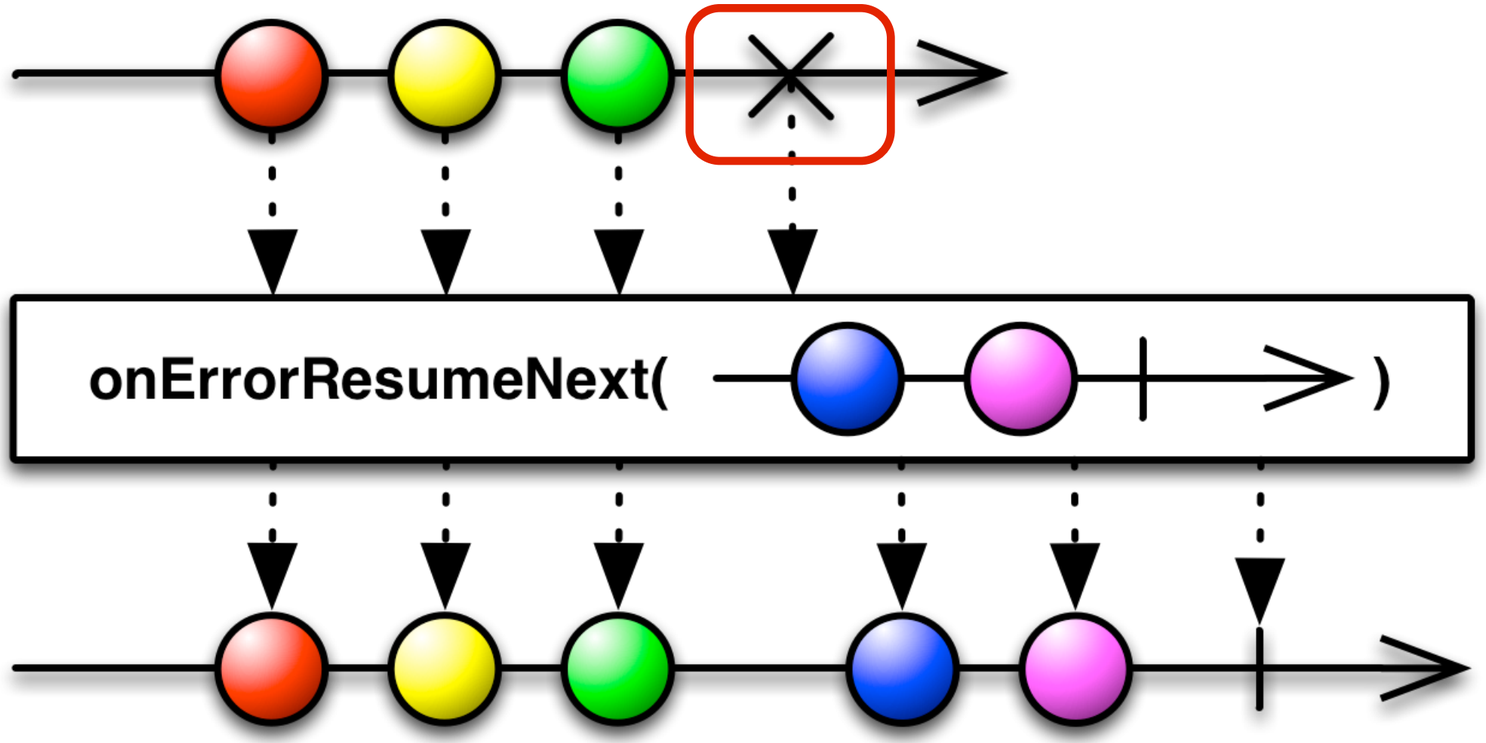

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                           .onErrorResumeNext(getFallbackForB());

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])},
        { exception -> println("error occurred: "
                               + exception.getMessage())})
```



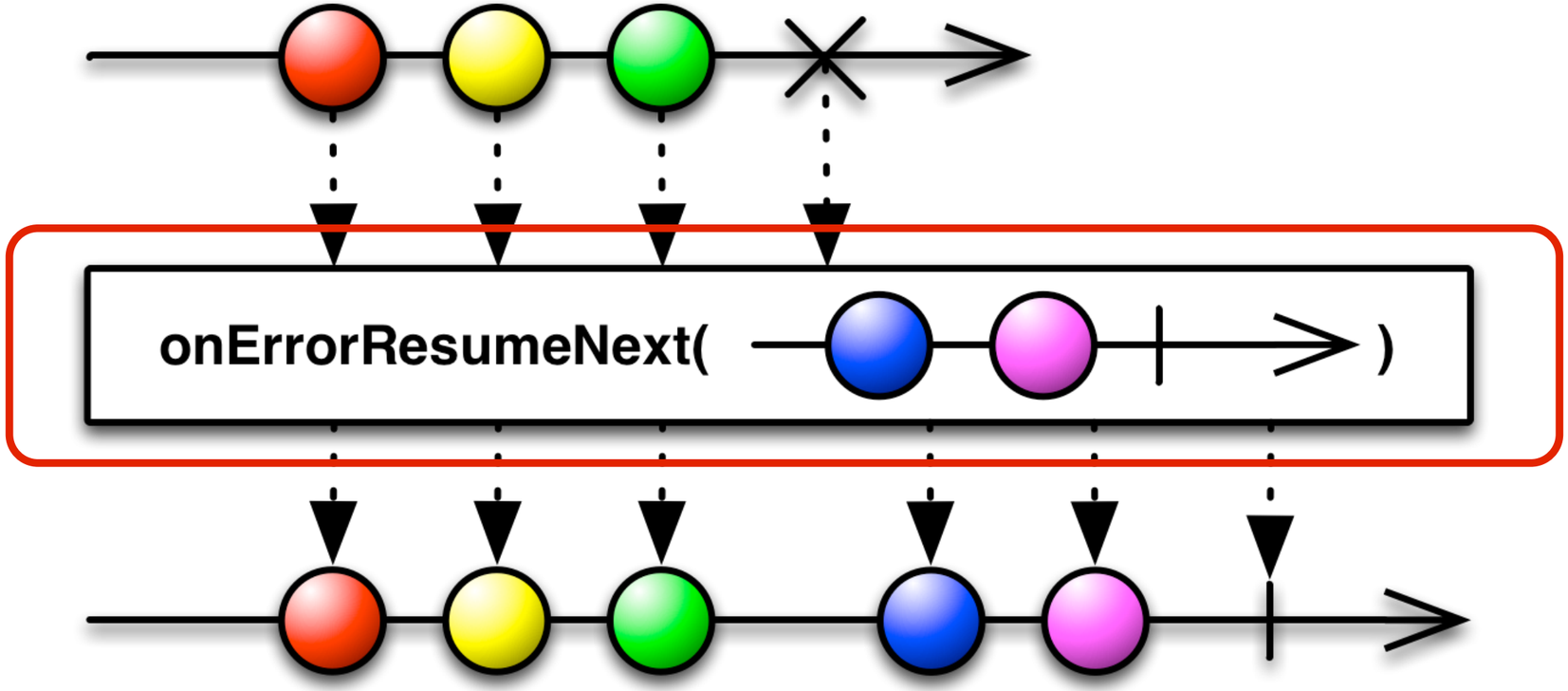

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                            .onErrorResumeNext(getFallbackForB());

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])},
        { exception -> println("error occurred: "
                               + exception.getMessage())})
```



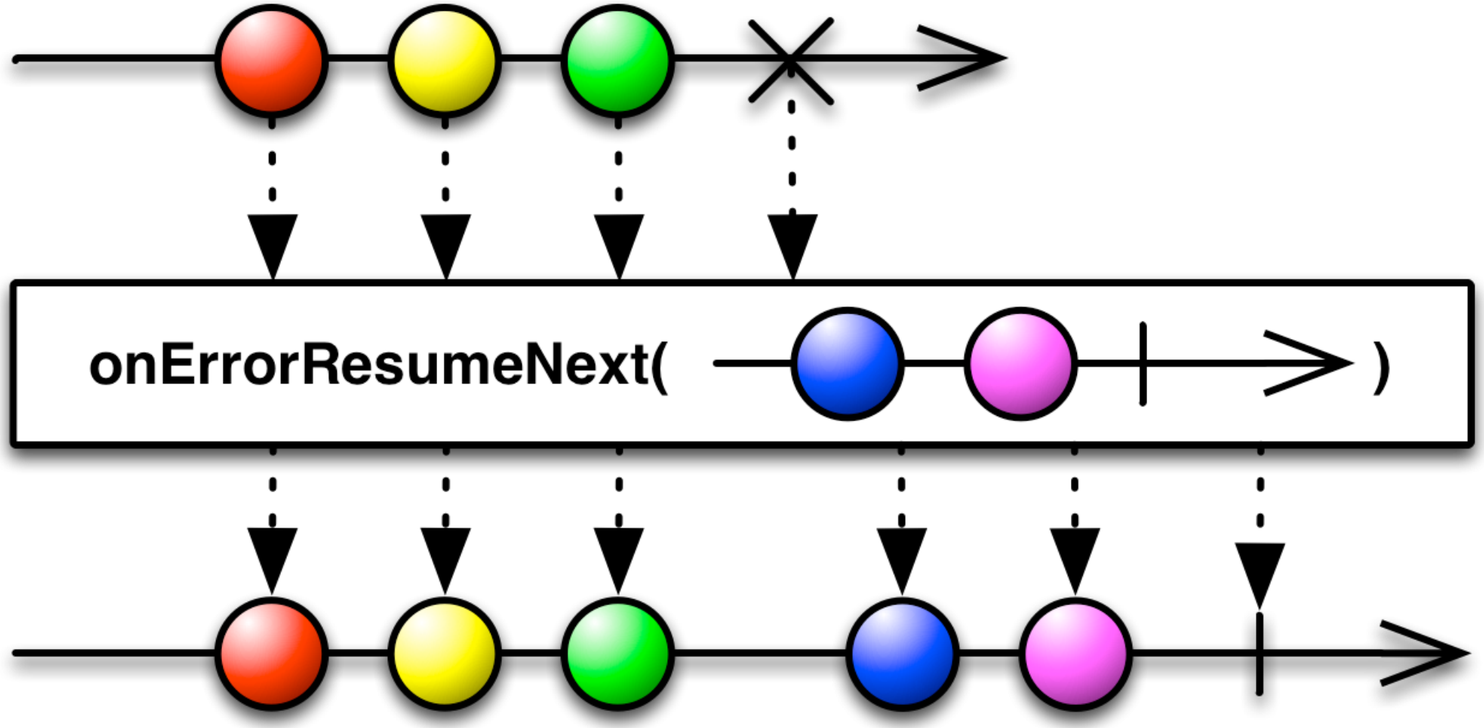

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                           .onErrorResumeNext(getFallbackForB());

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])},
        { exception -> println("error occurred: "
                               + exception.getMessage())})
```



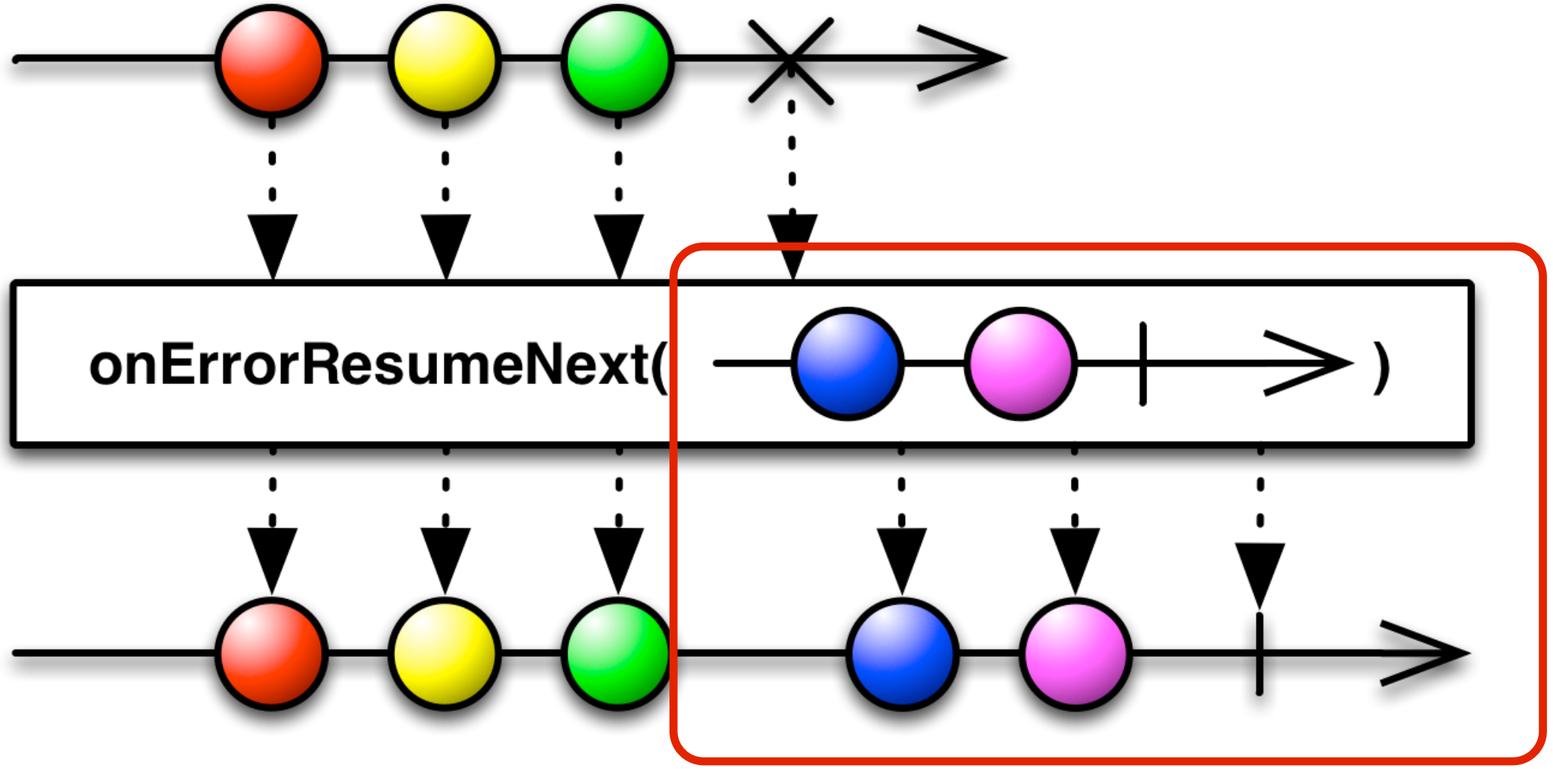

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                           .onErrorResumeNext(getFallbackForB());

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])},
        { exception -> println("error occurred: "
                               + exception.getMessage())})
```



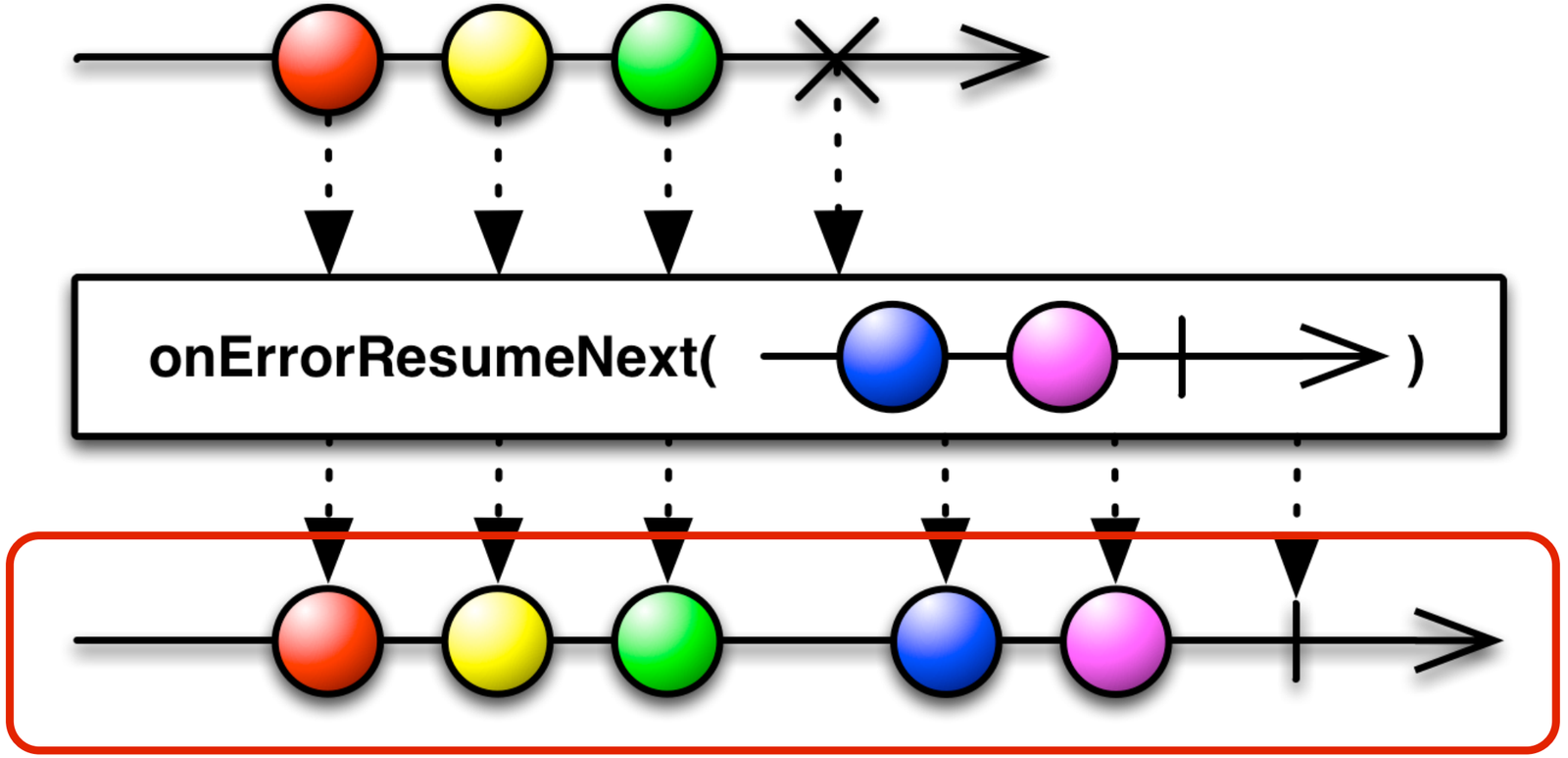

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                           .onErrorResumeNext(getFallbackForB());

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])},
        { exception -> println("error occurred: "
                               + exception.getMessage())})
```




```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                        .onErrorResumeNext(getFallbackForB());

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])},
        { exception -> println("error occurred: "
                                + exception.getMessage())})
```




```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                           .onErrorResumeNext(getFallbackForB());

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])},
        { exception -> println("error occurred: "
                               + exception.getMessage())})
```



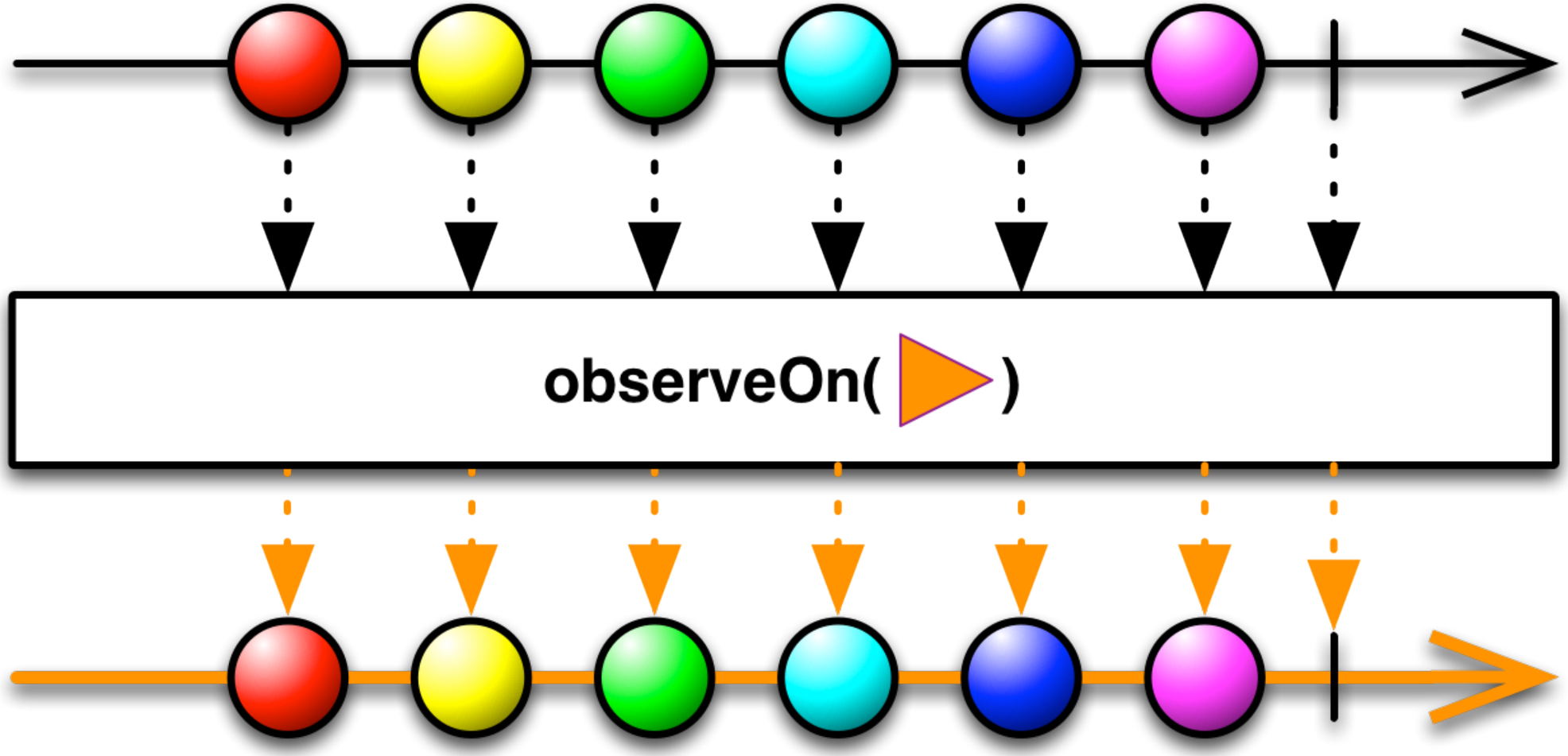

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                           .onErrorResumeNext(getFallbackForB());

Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])},
        { exception -> println("error occurred: "
                               + exception.getMessage())})
```

SCHEDULERS

```
public Observable<T> observeOn(Scheduler scheduler)
```

SCHEDULERS

```
public Observable<T> observeOn(Scheduler scheduler)
```



Schedulers

```
interval(long interval, TimeUnit unit)
interval(long interval, TimeUnit unit, Scheduler scheduler)
```

Schedulers

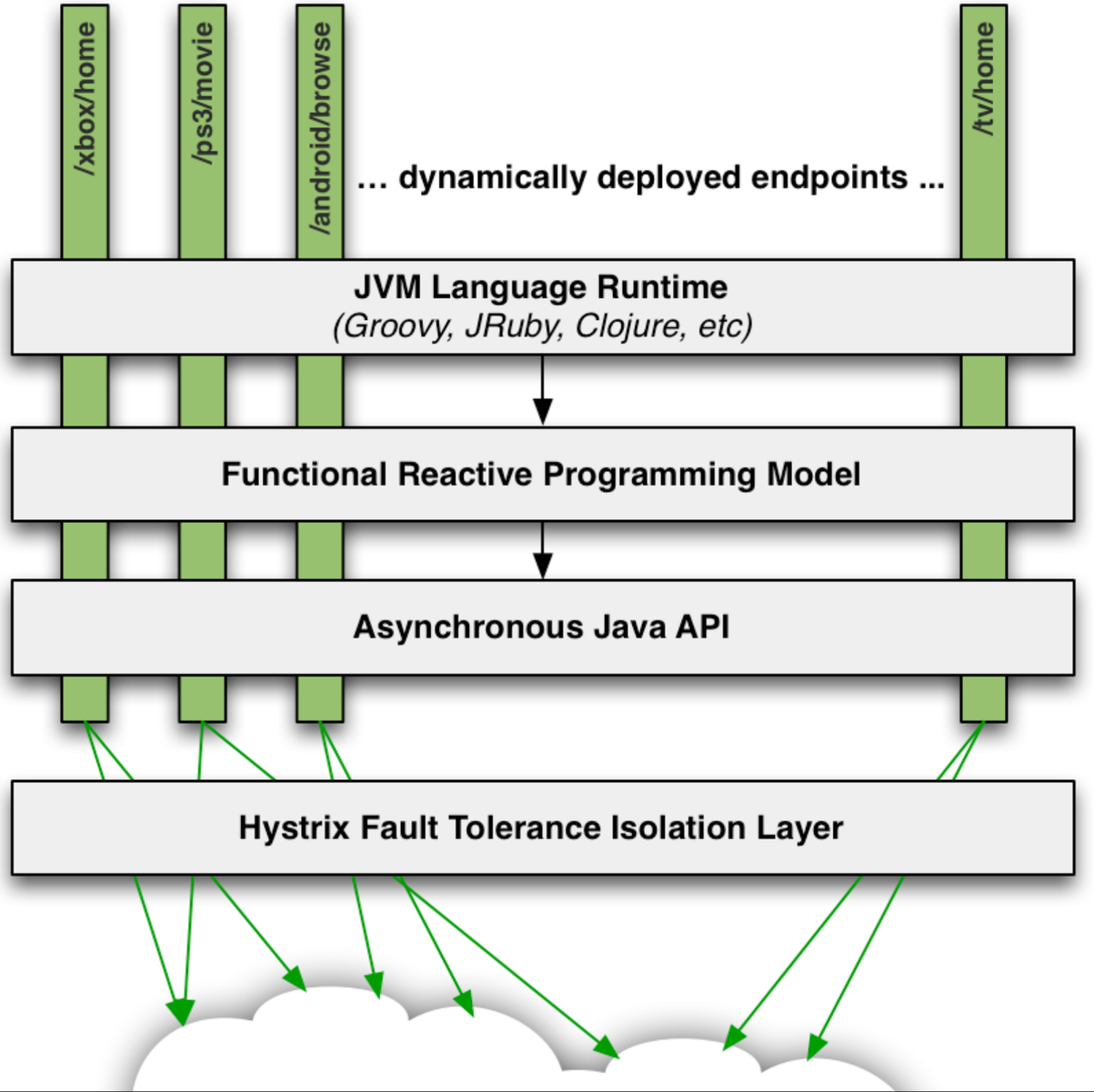
```
interval(long interval, TimeUnit unit)
interval(long interval, TimeUnit unit, Scheduler scheduler)
```

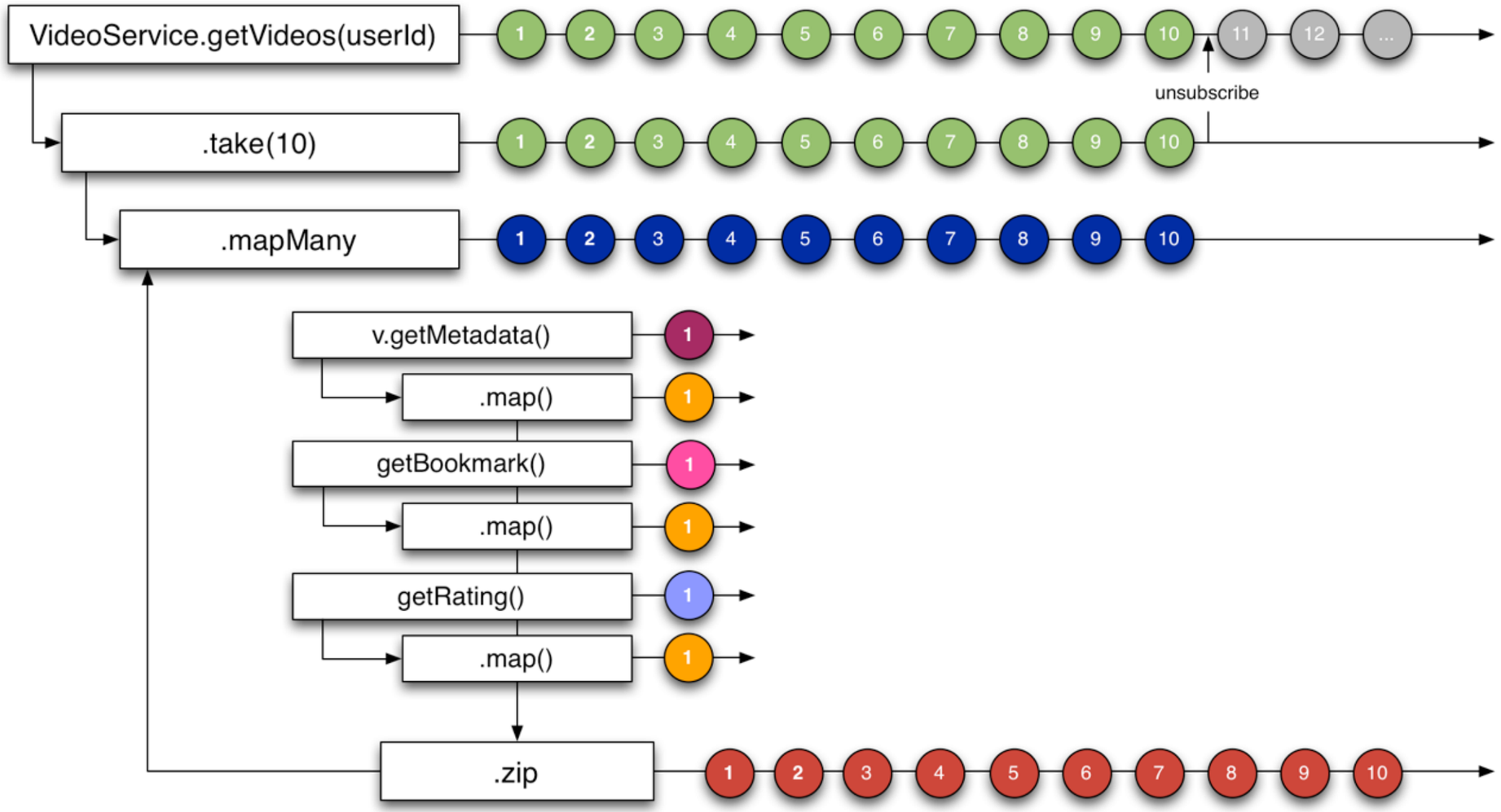
Schedulers

```
interval(long interval, TimeUnit unit)
interval(long interval, TimeUnit unit, Scheduler scheduler)
```

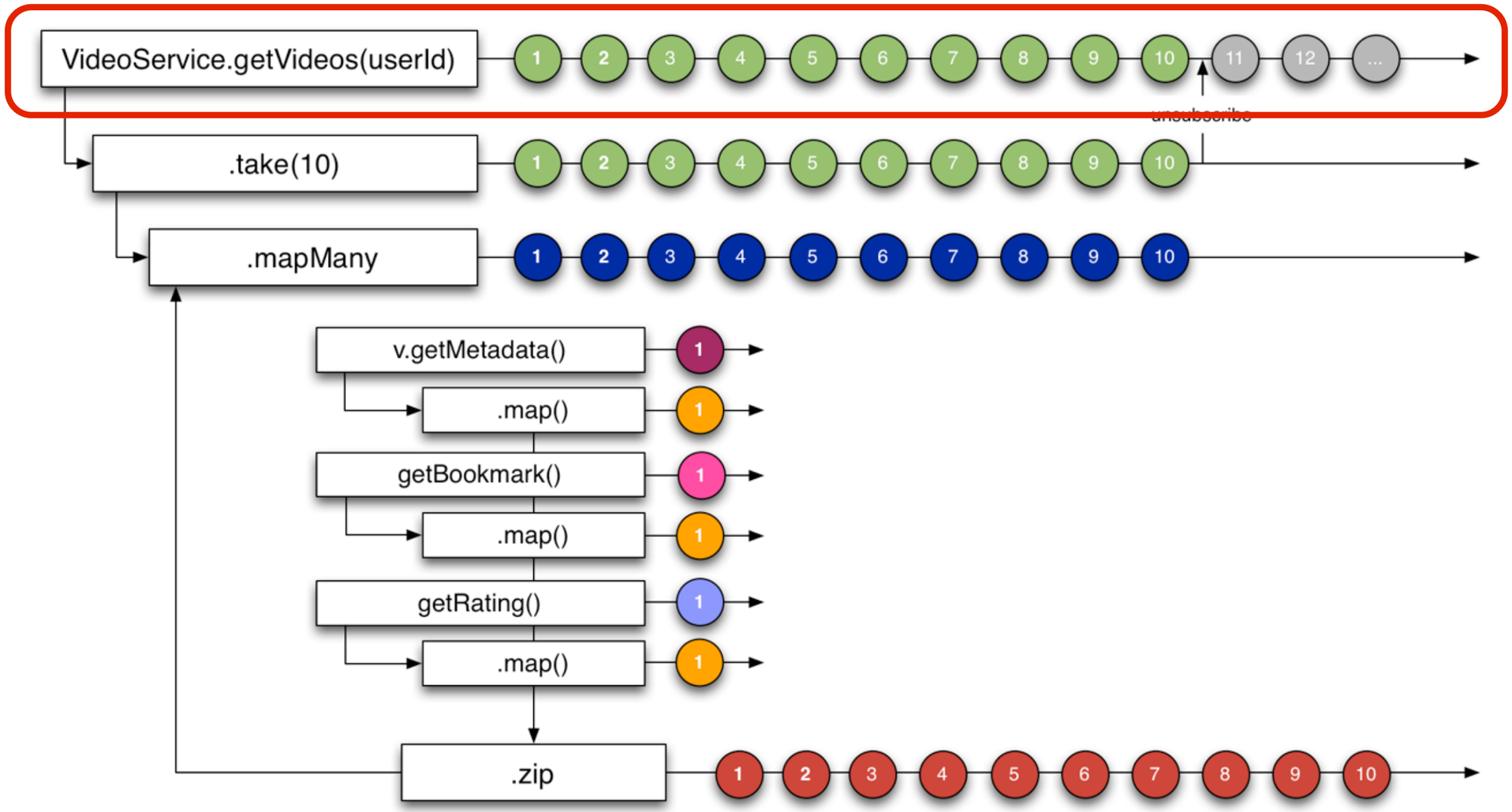
CurrentThreadScheduler
ExecutorScheduler
ImmediateScheduler
NewThreadScheduler
TestScheduler

Netflix API Use Case





[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]



[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

Observable<Video> emits n videos to onNext()

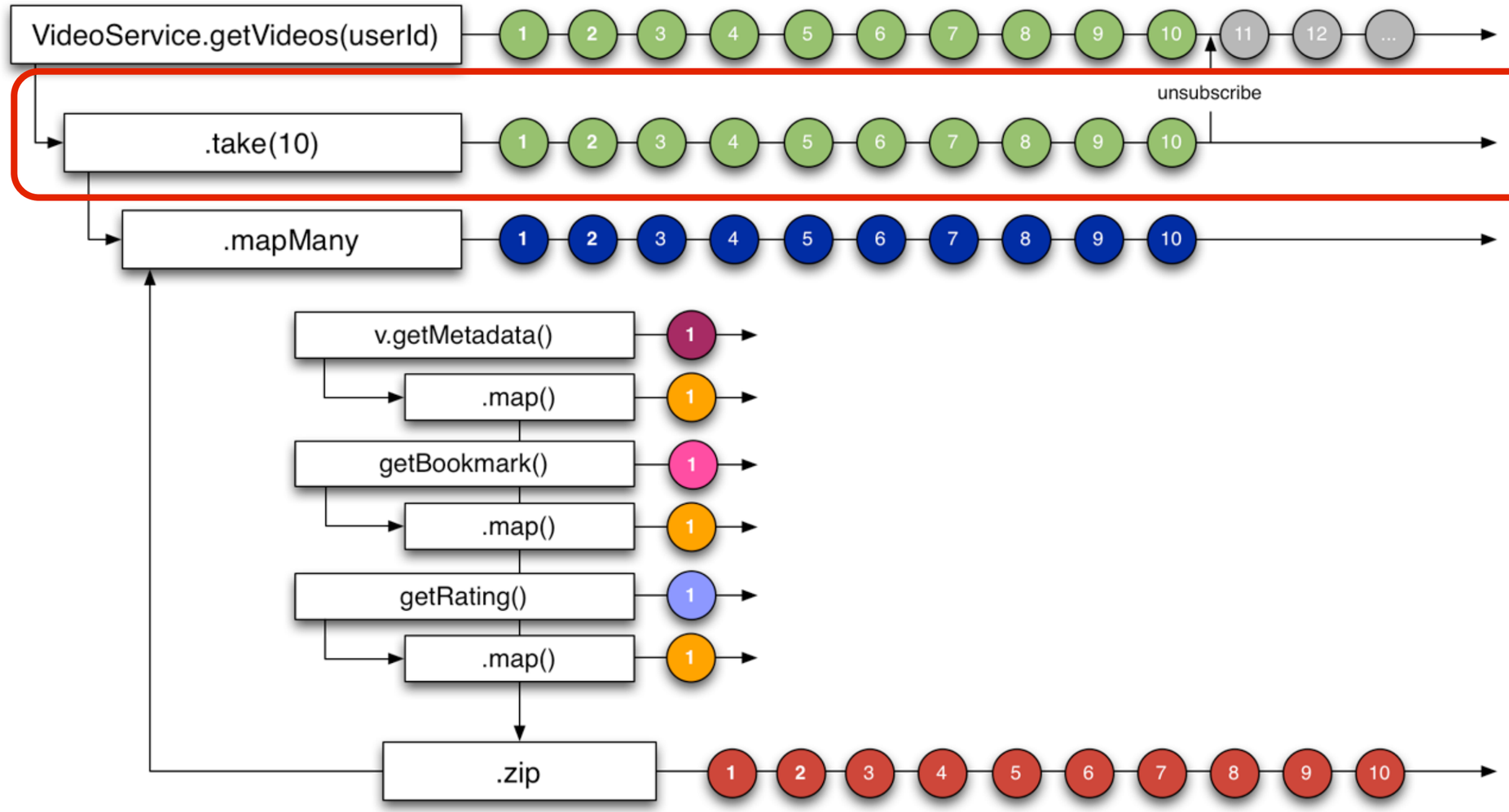
```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)

}
```

Observable<Video> emits n videos to onNext()

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
}
```

Takes first 10 then unsubscribes from origin.

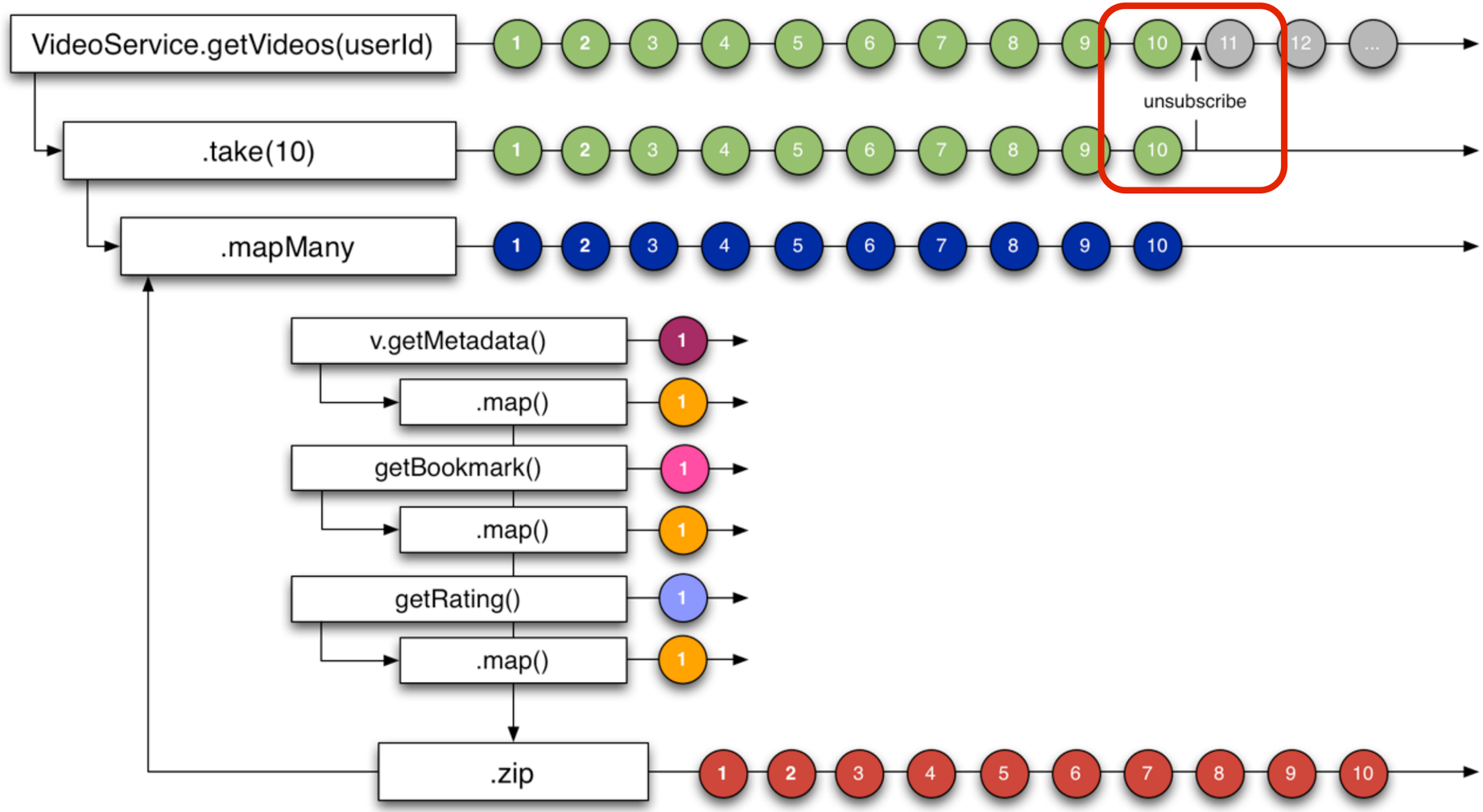
Returns Observable<Video> that emits 10 Videos.



[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

Takes first 10 then unsubscribes from origin.

Returns Observable<Video> that emits 10 Videos.



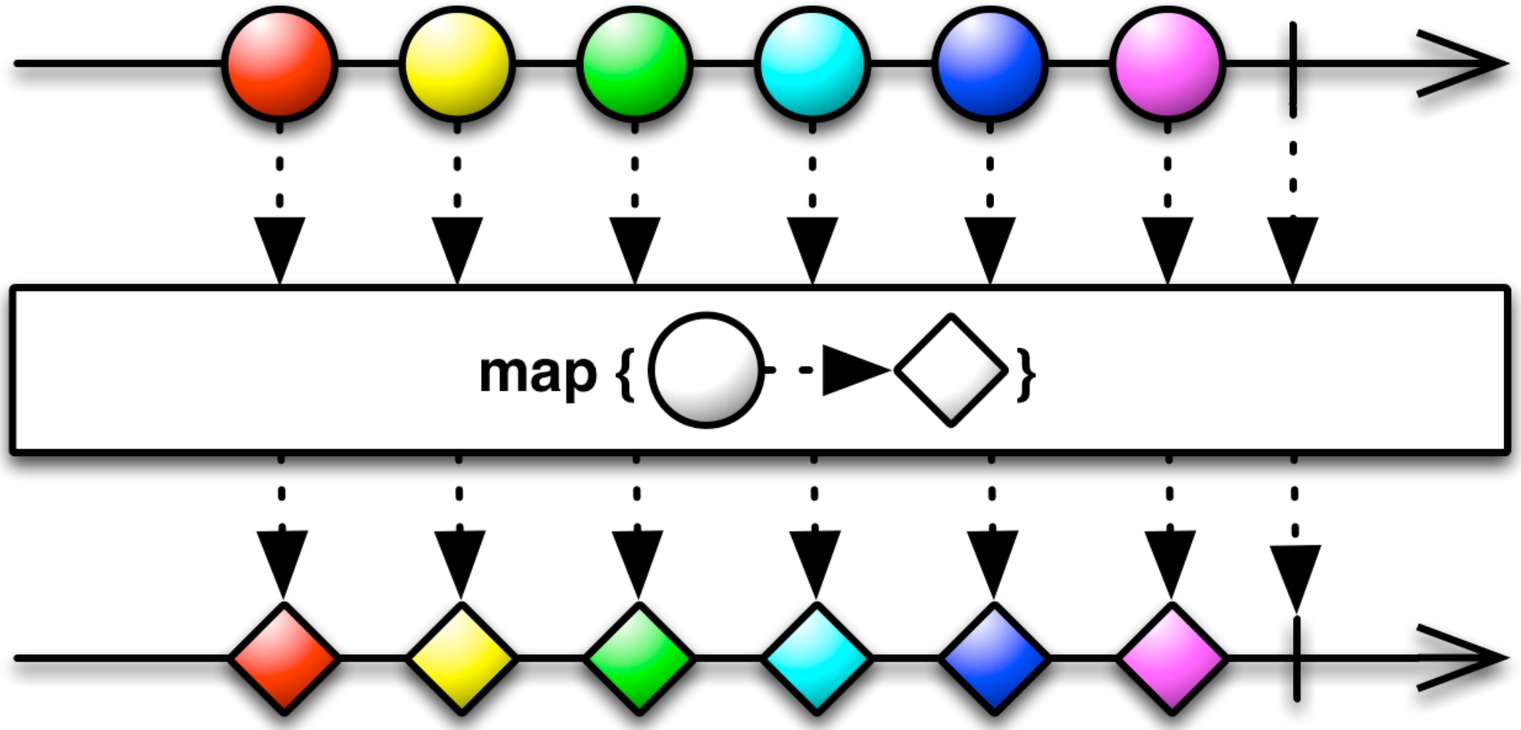
[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

Takes first 10 then unsubscribes from origin.

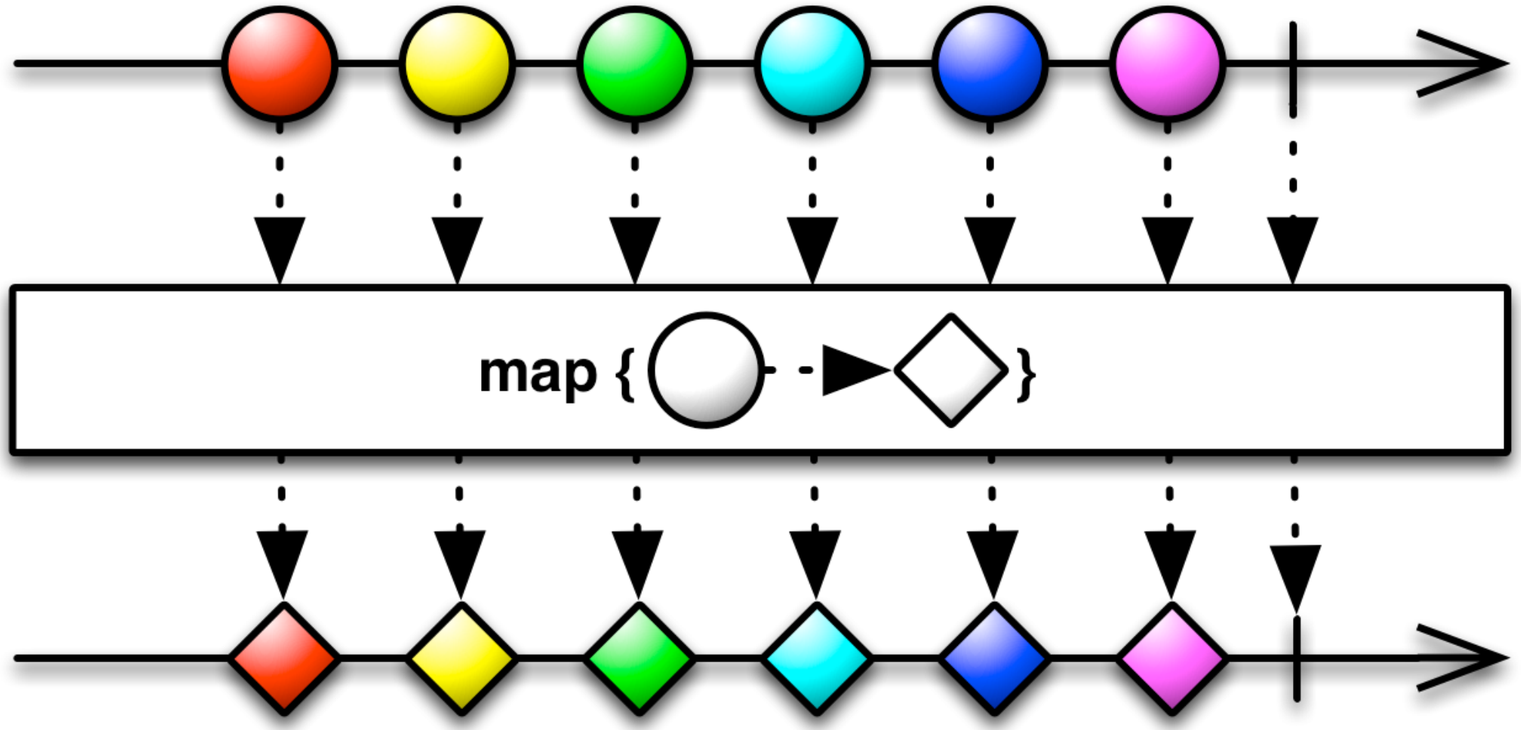
Returns Observable<Video> that emits 10 Videos.

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .map({ Video video ->
           // transform video object
        })
}
```

The ‘map’ operator allows transforming the input value into a different output.




```
Observable<R> b = Observable<T>.map({ T t ->
  R r = ... transform t ...
  return r;
})
```



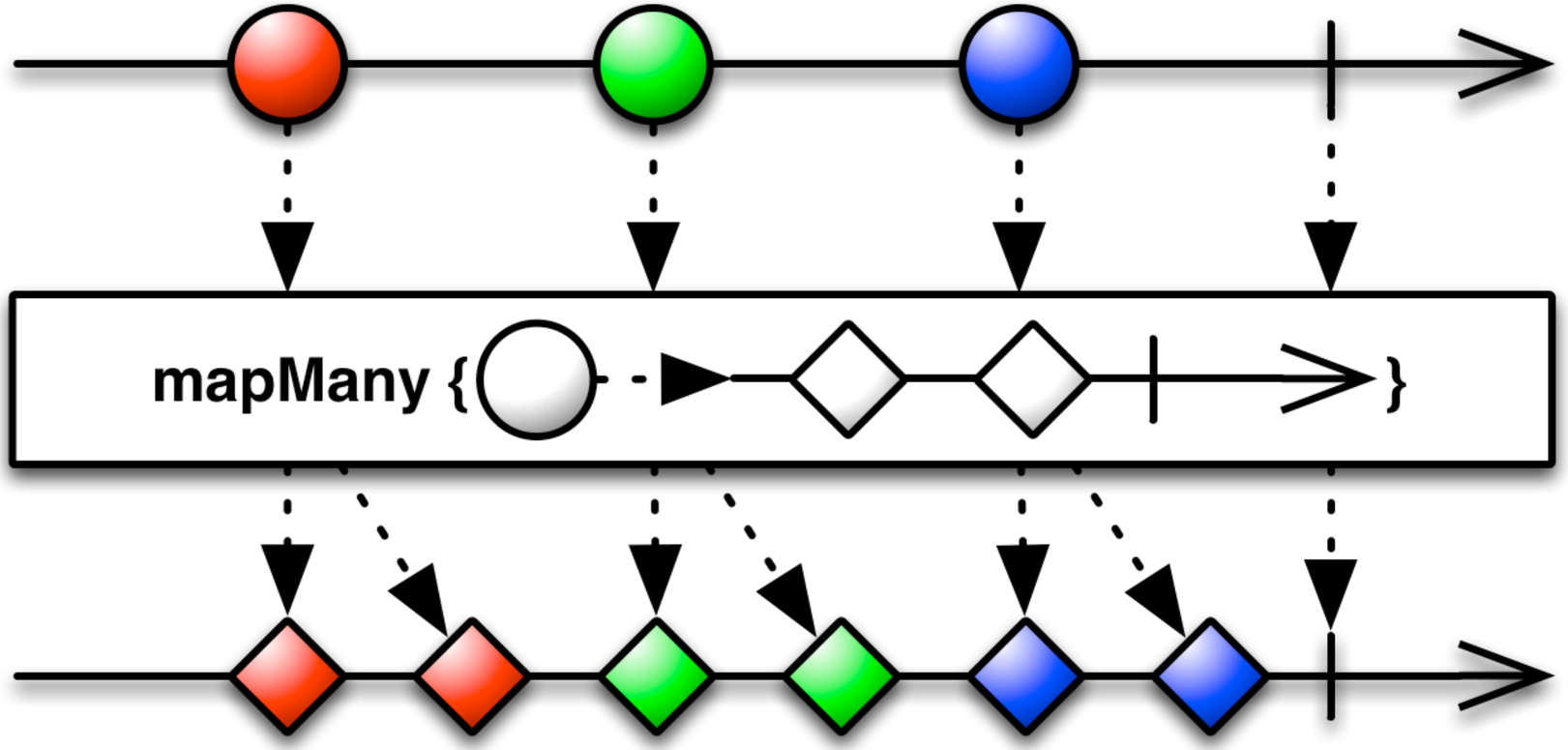

```
Observable<R> b = Observable<T>.map({ T t ->
  R r = ... transform t ...
  return r;
})
```

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .map({ Video video ->
           // transform video object
        })
}
```

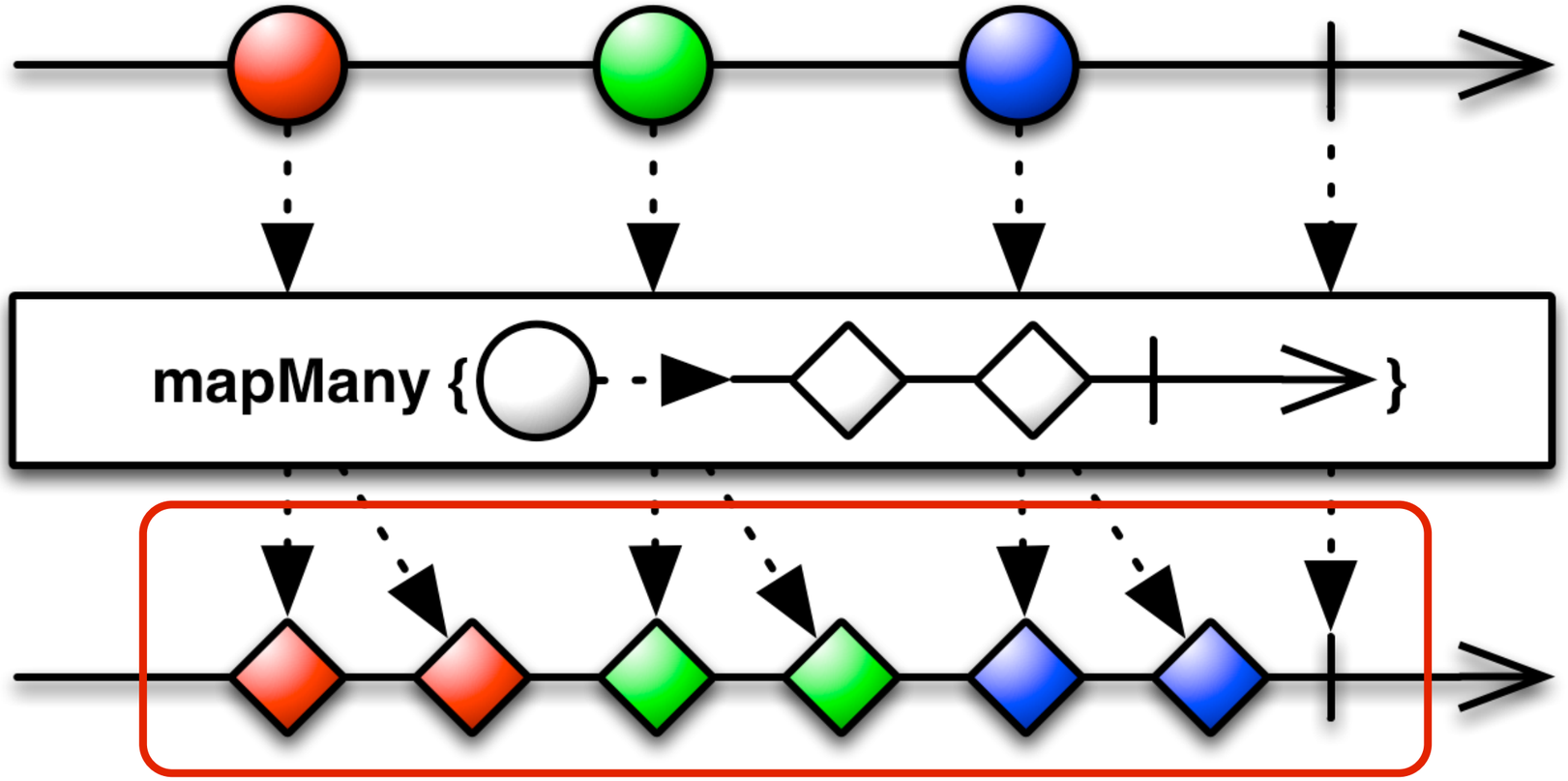
The ‘map’ operator allows transforming the input value into a different output.

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .flatMap( Video video ->
          // for each video we want to fetch metadata
          def m = video.getMetadata()
              .map({ Map<String, String> md ->
                // transform to the data and format we want
                return [title: md.get("title"),length: md.get("duration")]
           })
          // and its rating and bookmark
          def b ...
          def r ...
        })
}
```

We change to ‘mapMany’/‘flatMap’ which is like merge(map()) since we will return an Observable<T> instead of T.




```
                                                flatMap
Observable<R> b = Observable<T>.mapMany({ T t ->
    Observable<R> r = ... transform t ...
    return r;
})
```



flatMap

```
Observable<R> b = Observable<T>.mapMany({ T t ->
    Observable<R> r = ... transform t ...
    return r;
})
```

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .flatMap({ Video video ->
           // for each video we want to fetch metadata
          def m = video.getMetadata()
              .map({ Map<String, String> md ->
                // transform to the data and format we want
                return [title: md.get("title"),length: md.get("duration")]
           })
          // and its rating and bookmark
          def b ...
          def r ...
        })
}
```

Nested asynchronous calls
that return more Observables.

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .flatMap({ Video video ->
            // for each video we want to fetch metadata
          def m = video.getMetadata()
              .map({ Map<String, String> md ->
                // transform to the data and format we want
                return [title: md.get("title"),length: md.get("duration")]
            })
          // and its rating and bookmark
          def b ...
          def r ...
        })
}
```

Nested asynchronous calls
that return more Observables.

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .flatMap({ Video video ->
            // for each video we want to fetch metadata
            def m = video.getMetadata()
                .map({ Map<String, String> md ->
                    // transform to the data and format we want
                    return [title: md.get("title"),length: md.get("duration")]
                })
            // and its rating and bookmark
            def b ...
            def r ...
        })
}
```

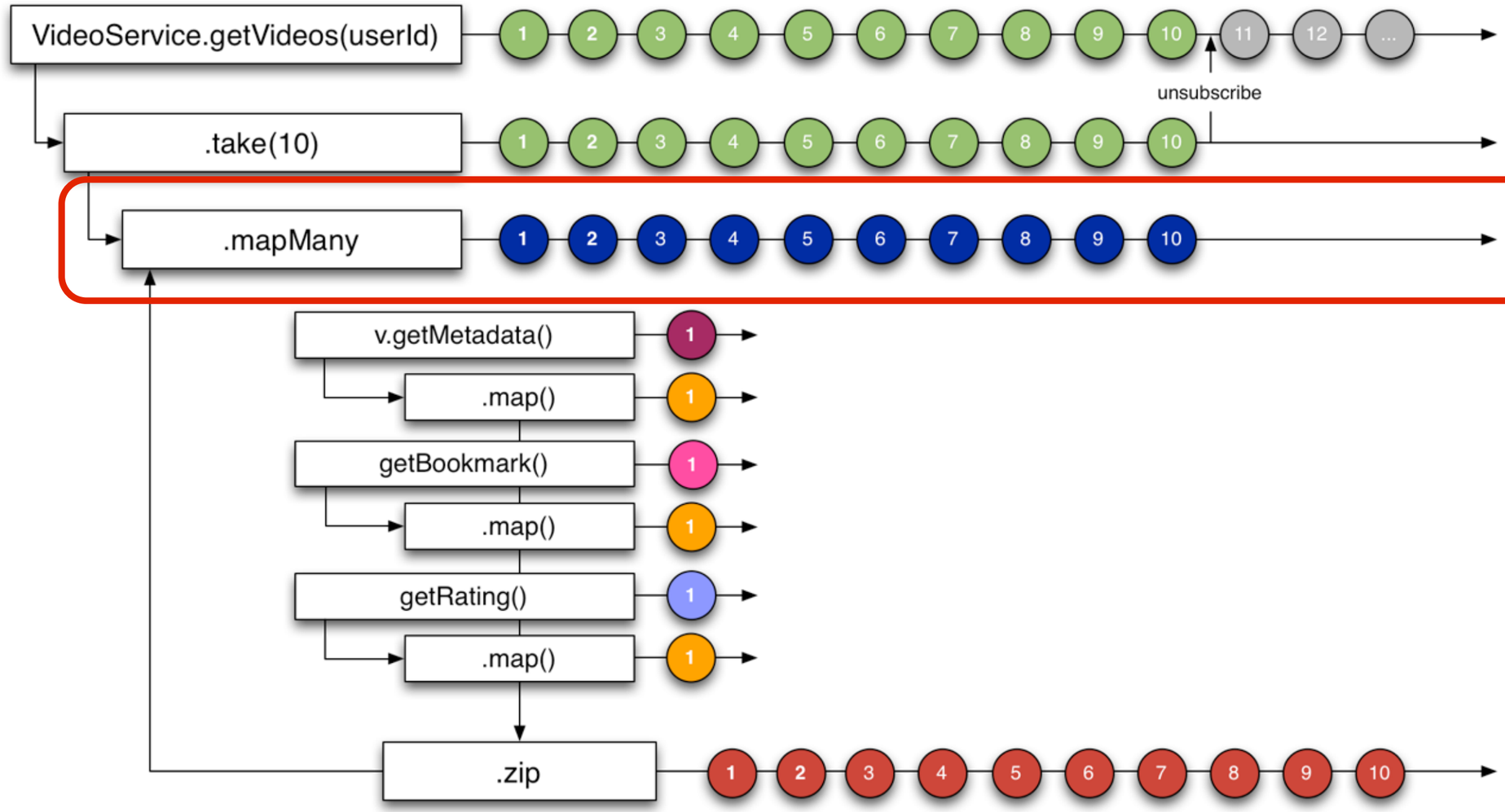
Observable<VideoMetadata>

Observable<VideoBookmark>

Observable<VideoRating>

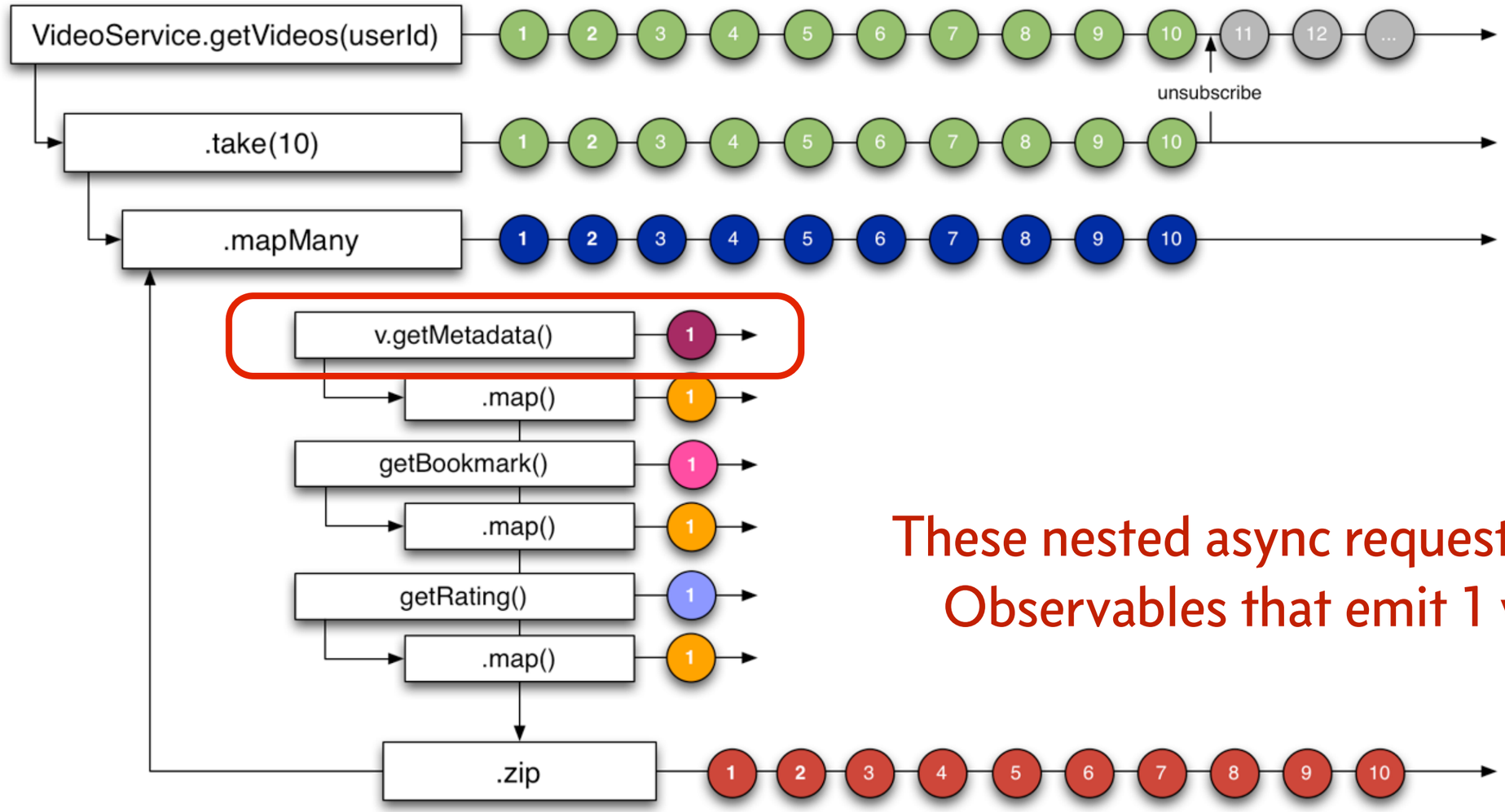
```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .flatMap({ Video video ->
           // for each video we want to fetch metadata
          def m = video.getMetadata()
                .map({ Map<String, String> md ->
                  // transform to the data and format we want
                  return [title: md.get("title"),length: md.get("duration")]
           })
          // and its rating and bookmark
          def b ...
          def r ...
        })
}
```

Each Observable transforms
its data using ‘map’



[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

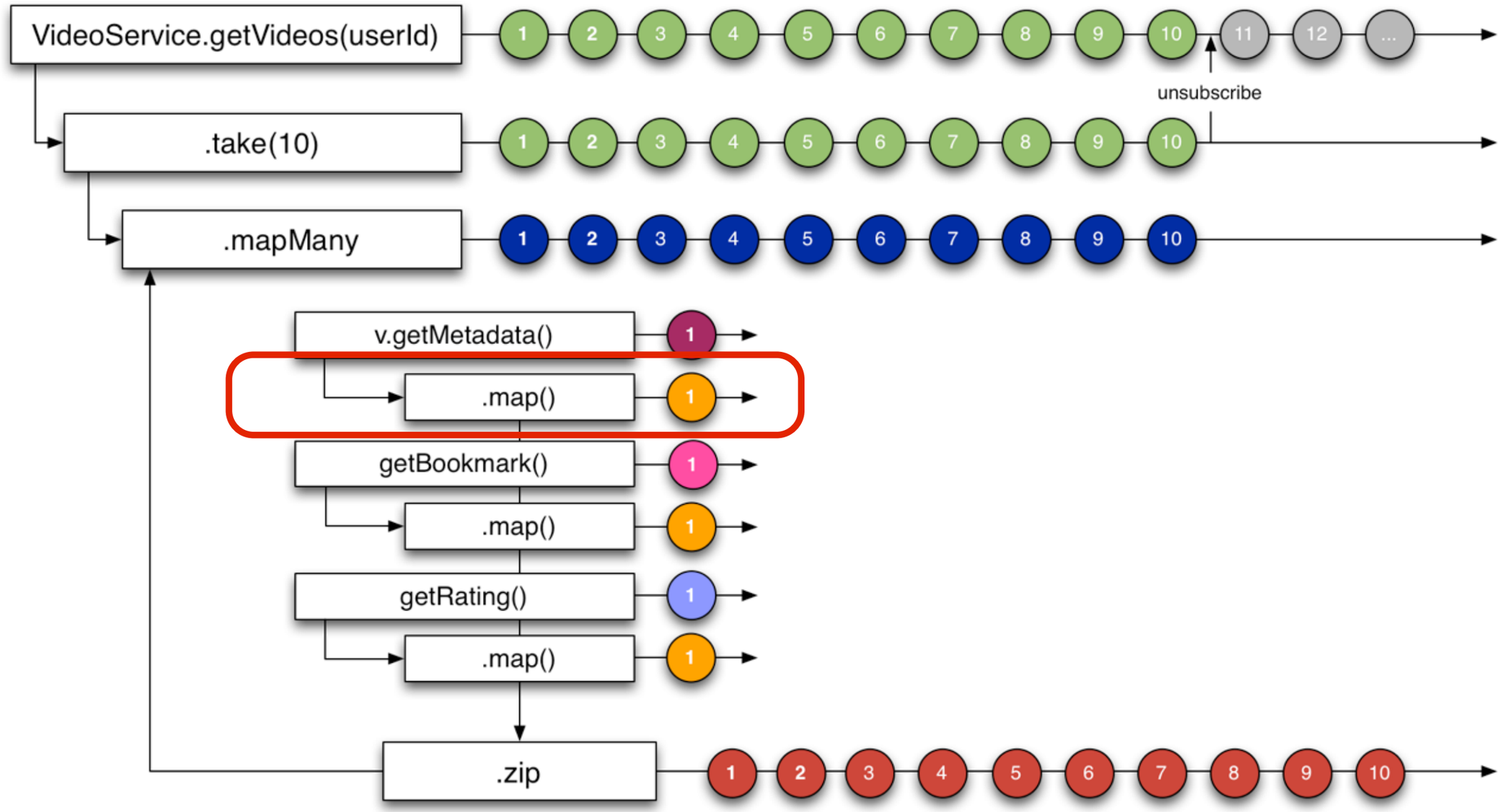
For each of the 10 Video objects it transforms
via ‘mapMany’ function that does nested async calls.



These nested async requests return Observables that emit 1 value.

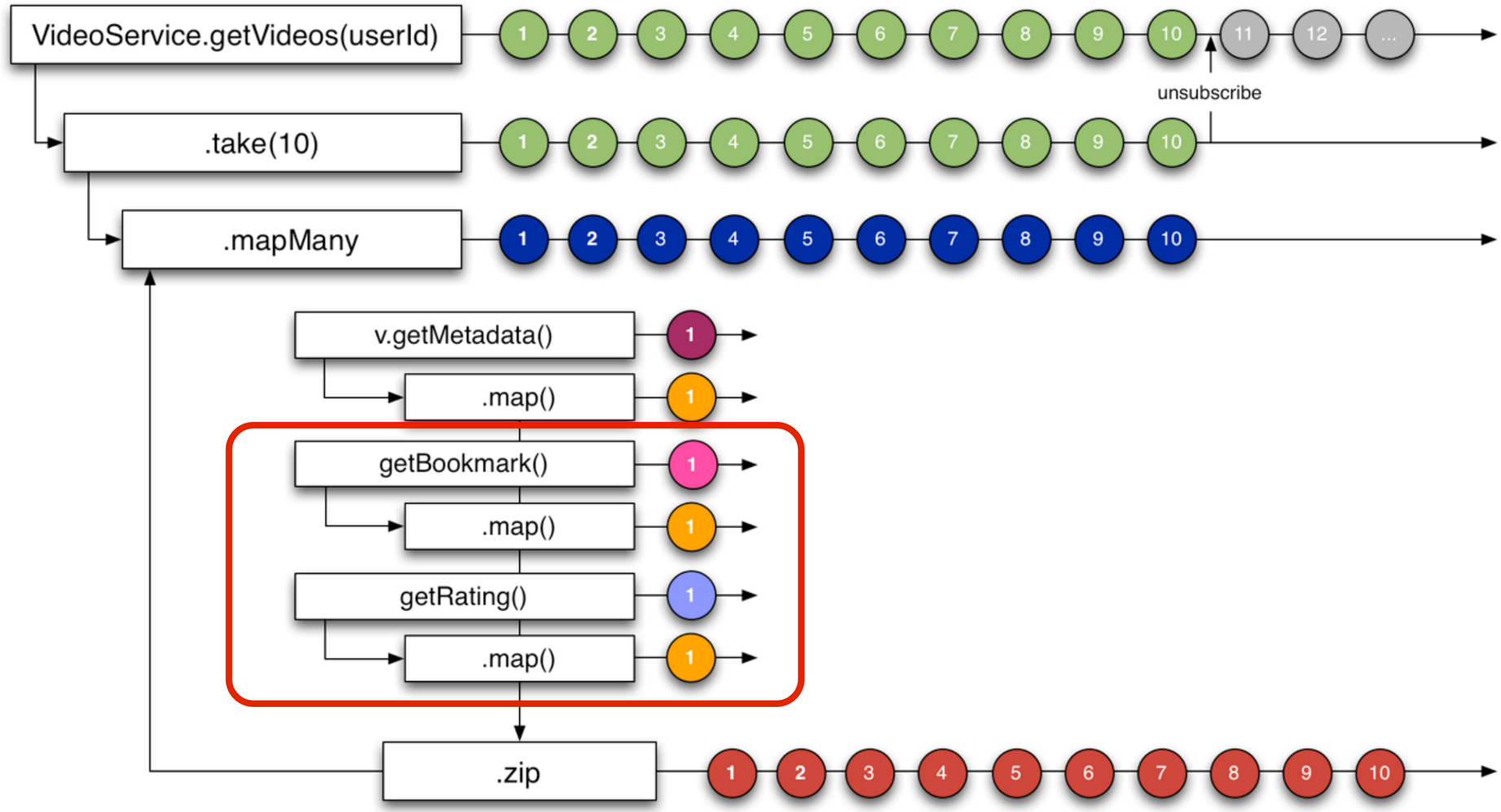
[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

For each Video ‘v’ it calls getMetadata()
which returns Observable<VideoMetadata>



[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

The Observable<VideoMetadata> is transformed via a 'map' function to return a Map of key/values.



[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

Same for Observable<VideoBookmark> and Observable<VideoRating>

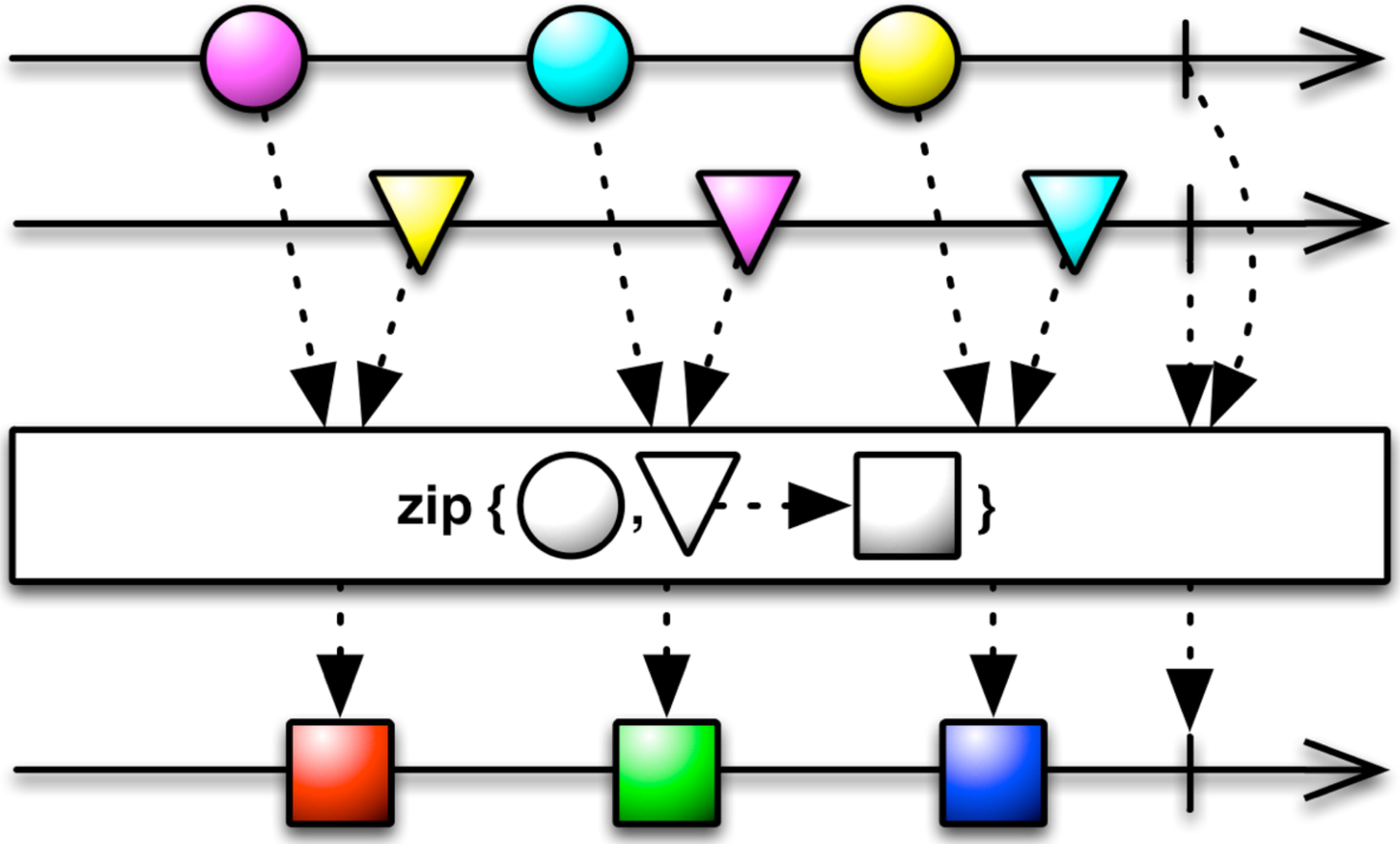
```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
       .take(10)
       .flatMap({ Video video ->
          // for each video we want to fetch metadata
          def m = video.getMetadata()
             .map({ Map<String, String> md ->
               // transform to the data and format we want
               return [title: md.get("title"),length: md.get("duration")]
          })
         // and its rating and bookmark
         def b ...
         def r ...
         // compose these together
       })
}
```

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .flatMap({ Video video ->
          def m ...
          def b ...
          def r ...
          // compose these together
    })
}
```

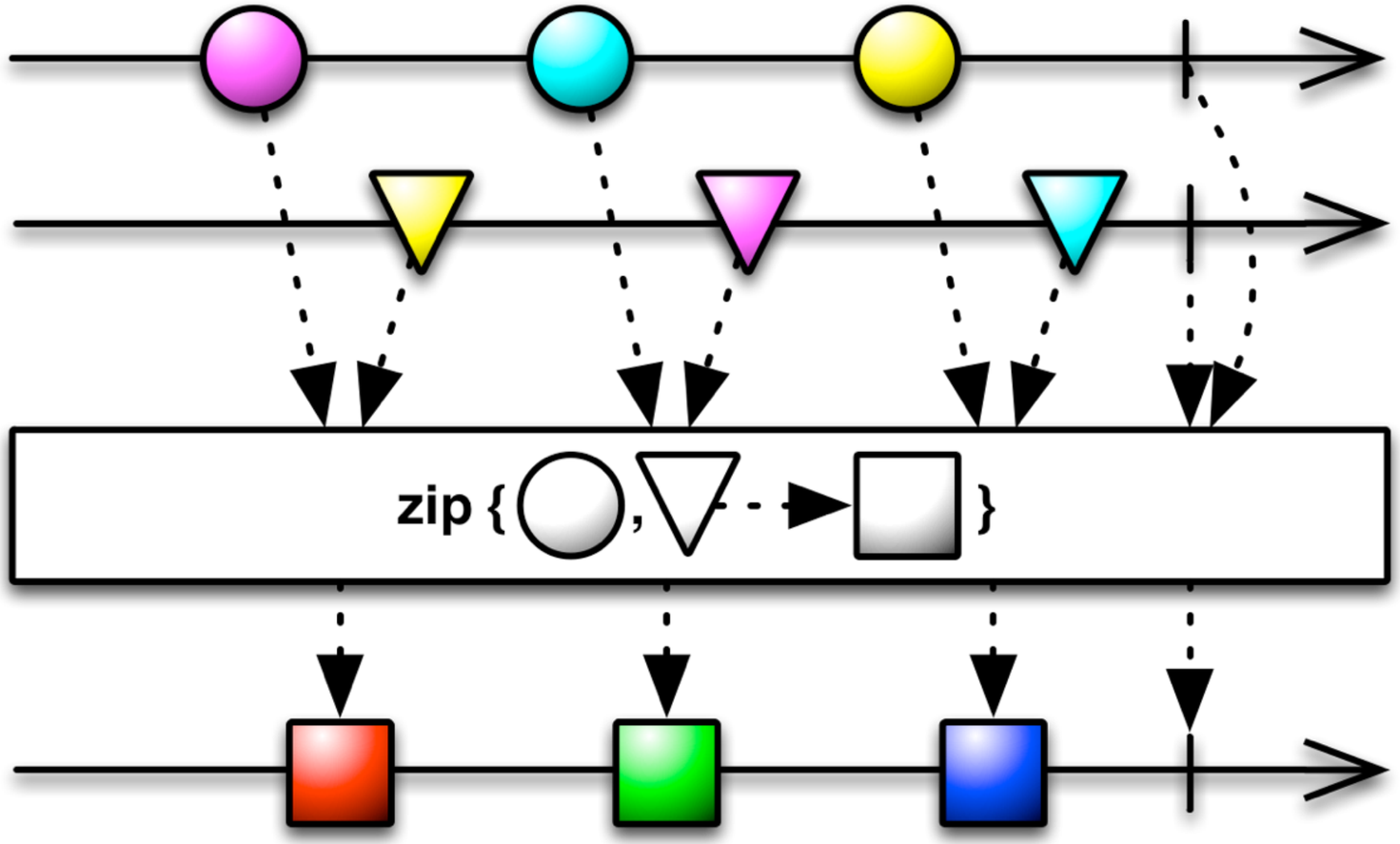
```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .flatMap({ Video video ->
          def m ...
          def b ...
          def r ...
          // compose these together
          return Observable.zip(m, b, r, { metadata, bookmark, rating ->
              // now transform to complete dictionary
              // of data we want for each Video
              return [id: video.videoId] << metadata << bookmark << rating
          })
    })
}
```

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .flatMap({ Video video ->
          def m ...
          def b ...
          def r ...
          // compose these together
          return Observable.zip(m, b, r, { metadata, bookmark, rating ->
              // now transform to complete dictionary
              // of data we want for each Video
              return [id: video.videoId] << metadata << bookmark << rating
          })
    })
}
```

The ‘zip’ operator combines the 3 asynchronous Observables into 1




```
Observable.zip(a, b, { a, b, ->
  ... operate on values from both a & b ...
  return [a, b]; // i.e. return tuple
})
```



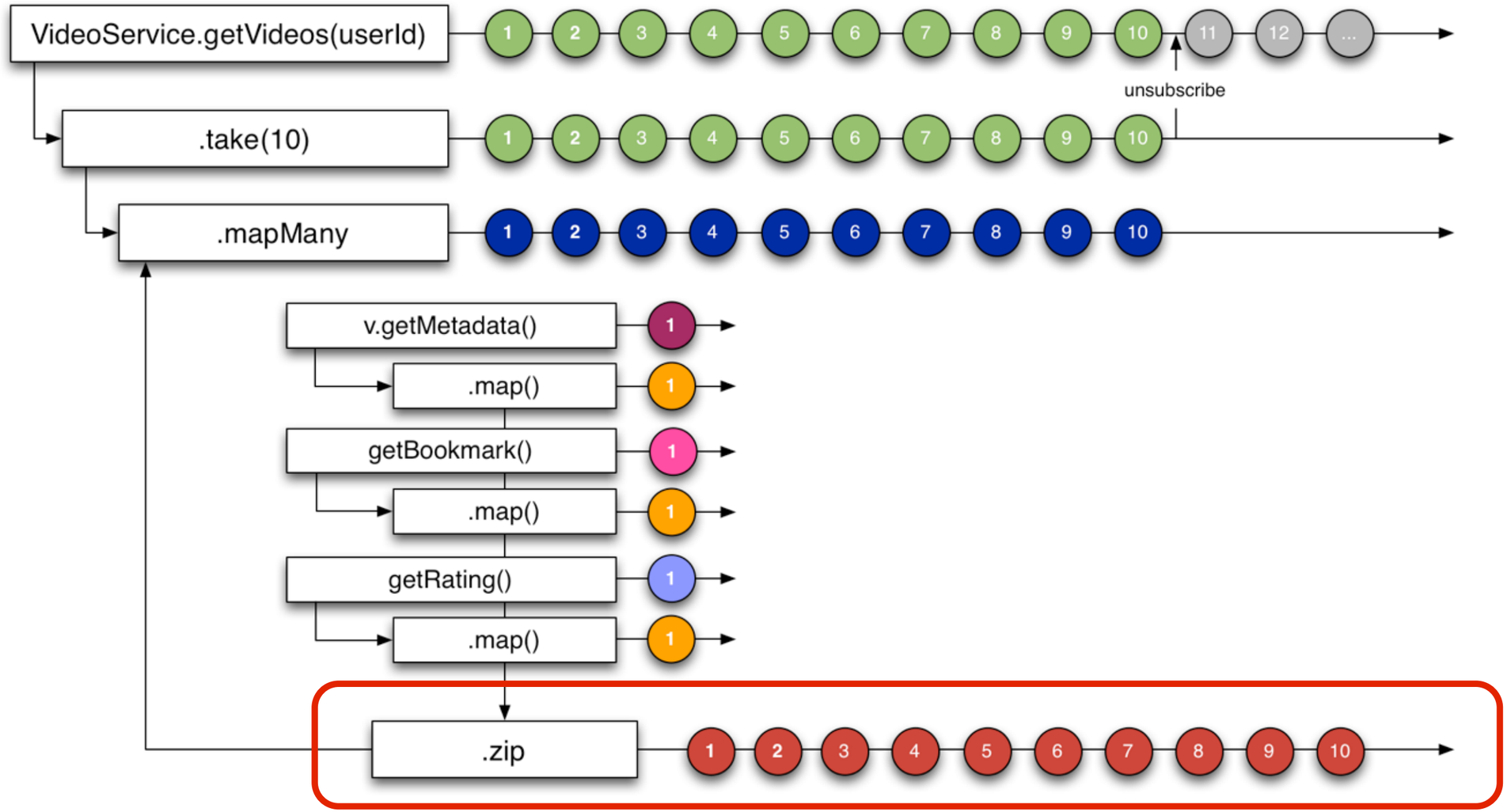

```
Observable.zip(a, b, { a, b, ->
    ... operate on values from both a & b ...
    return [a, b]; // i.e. return tuple
})
```

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .flatMap({ Video video ->
          def m ...
          def b ...
          def r ...
          // compose these together
          return Observable.zip(m, b, r, { metadata, bookmark, rating ->
              // now transform to complete dictionary
              // of data we want for each Video
              return [id: video.videoId] << metadata << bookmark << rating
          })
    })
}
```

return a single Map (dictionary) of transformed and combined data from 4 asynchronous calls

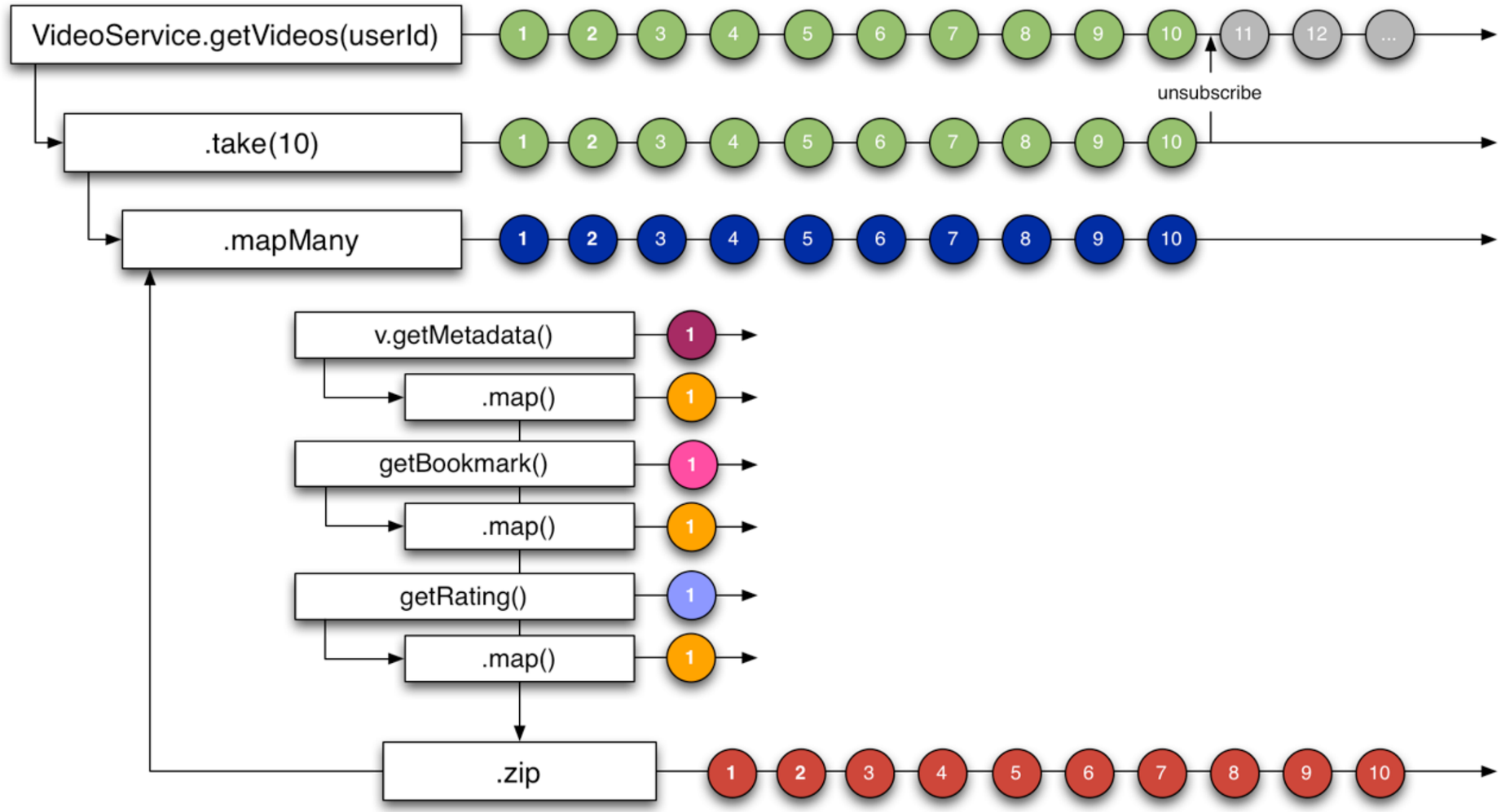
```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .flatMap({ Video video ->
          def m ...
          def b ...
          def r ...
          // compose these together
          return Observable.zip(m, b, r, { metadata, bookmark, rating ->
              // now transform to complete dictionary
              // of data we want for each Video
              return [id: video.videoId] << metadata << bookmark << rating
          })
    })
}
```

return a single Map (dictionary) of transformed and combined data from 4 asynchronous calls



[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

The ‘mapped’ Observables are combined with a ‘zip’ function that emits a Map (dictionary) with all data.



[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

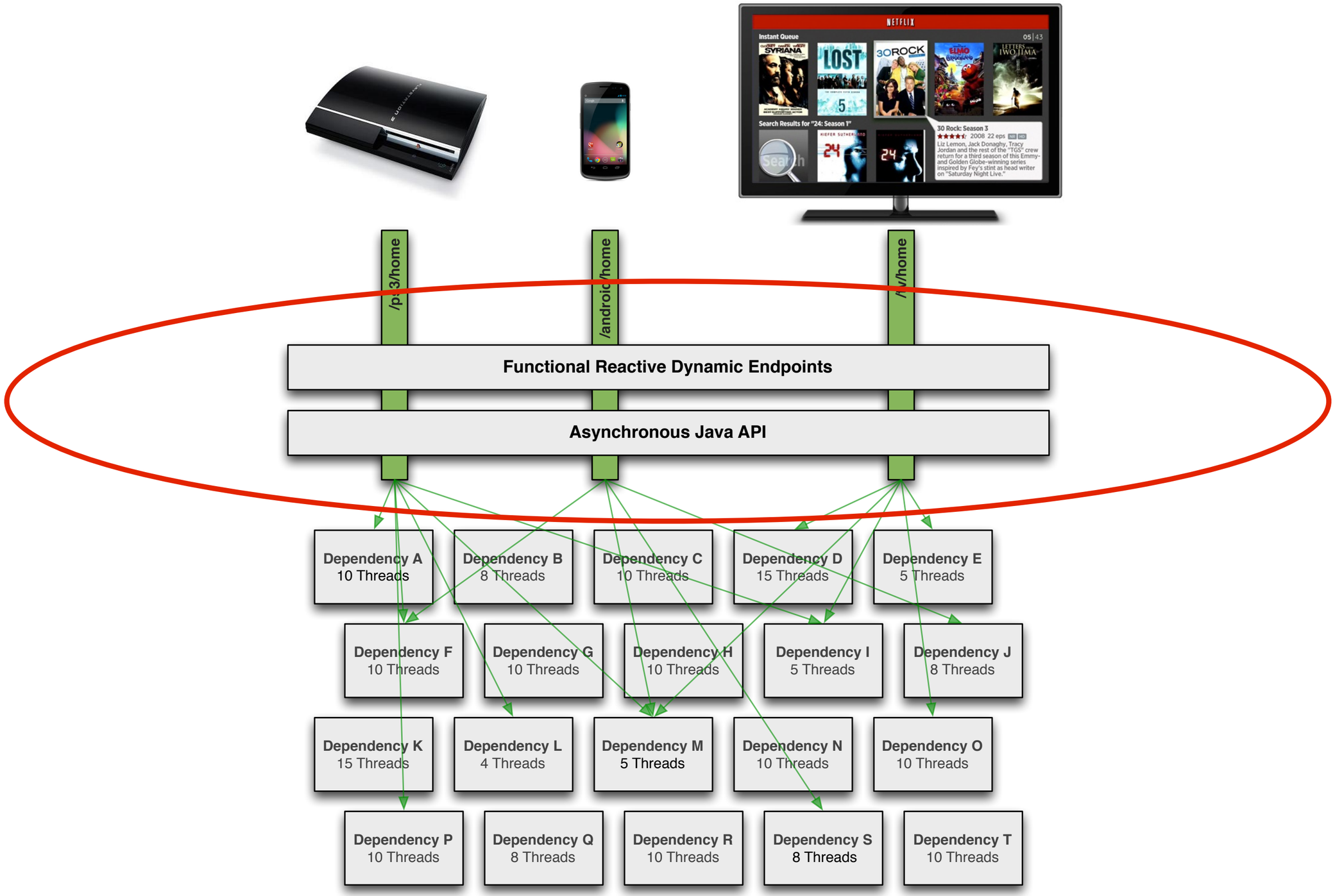
The full sequence returns Observable<Map> that emits a Map (dictionary) for each of 10 Videos.

INTERACTIONS WITH THE API

ARE ASYNCHRONOUS AND DECLARATIVE

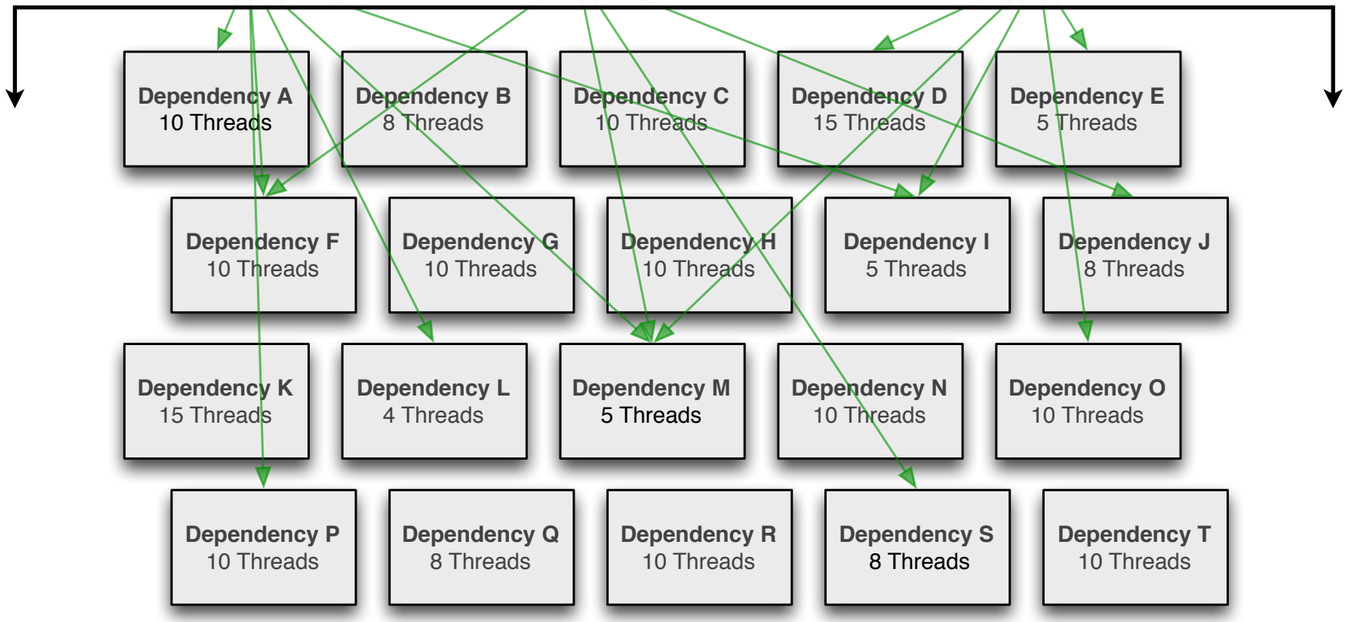
API IMPLEMENTATION CONTROLS

CONCURRENCY BEHAVIOR


NETFLIX
Instant Queue
05 | 43
Search Results for "24: Season 1"
30 Rock: Season 3
2008 22 eps
Liz Lemon, Jack Donaghy, Tracy Jordan and the rest of the "TGS" crew return for a third season of this Emmy- and Golden Globe-winning series inspired by Fey's stint as head writer on "Saturday Night Live."
/ps3/home
/android/home
/tv/home
Functional Reactive Dynamic Endpoints
Asynchronous Java API
Dependency A
10 Threads
Dependency B
8 Threads
Dependency C
10 Threads
Dependency D
15 Threads
Dependency E
5 Threads
Dependency F
10 Threads
Dependency G
10 Threads
Dependency H
10 Threads
Dependency I
5 Threads
Dependency J
8 Threads
Dependency K
15 Threads
Dependency L
4 Threads
Dependency M
5 Threads
Dependency N
10 Threads
Dependency O
10 Threads
Dependency P
10 Threads
Dependency Q
8 Threads
Dependency R
10 Threads
Dependency S
8 Threads
Dependency T
10 Threads

HYSTRIX

FAULT-ISOLATION LAYER

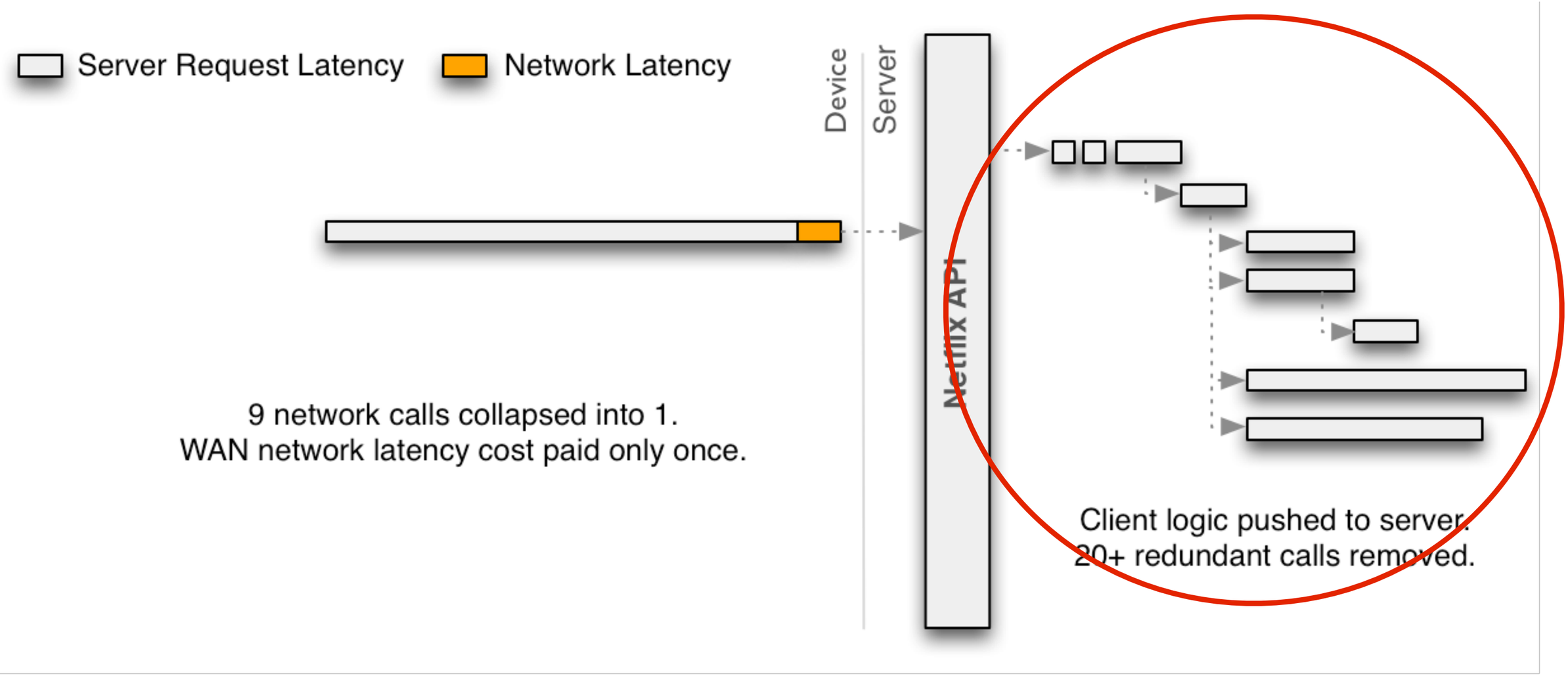


```
Observable<User> u = new GetUserCommand(id).observe();
Observable<Geo> g = new GetGeoCommand(request).observe();

Observable.zip(u, g, {user, geo ->
        return [username: user.getUsername(),
                currentLocation: geo.getCounty()]
})
```

RxJava in Hystrix 1.3+

https://github.com/Netflix/Hystrix

OBSERVABLE APIs



LESSONS LEARNED

DEVELOPER TRAINING & DOCUMENTATION

LESSONS LEARNED

DEVELOPER TRAINING & DOCUMENTATION

DEBUGGING AND TRACING

LESSONS LEARNED

Developer Training & Documentation

Debugging and Tracing

Only "rule" has been "don't mutate state outside of function"

ASYNCHRONOUS

VALUES

EVENTS

PUSH

FUNCTIONAL REACTIVE

LAMBDAS

CLOSURES

(MOSTLY) PURE

COMPOSABLE

jobs.netflix.com

Functional Reactive in the Netflix API with RxJava

http://techblog.netflix.com/2013/02/rxjava-netflix-api.html

Optimizing the Netflix API

http://techblog.netflix.com/2013/01/optimizing-netflix-api.html

RxJava

https://github.com/Netflix/RxJava

@RxJava

RxJS

http://reactive-extensions.github.io/RxJS/

@ReactiveX

Ben Christensen

@benjchristensen

http://www.linkedin.com/in/benjchristensen