Functional Reactive Programming with RxJava

GOTO Aarhus - October 2013

Ben Christensen

Software Engineer – Edge Platform at Netflix

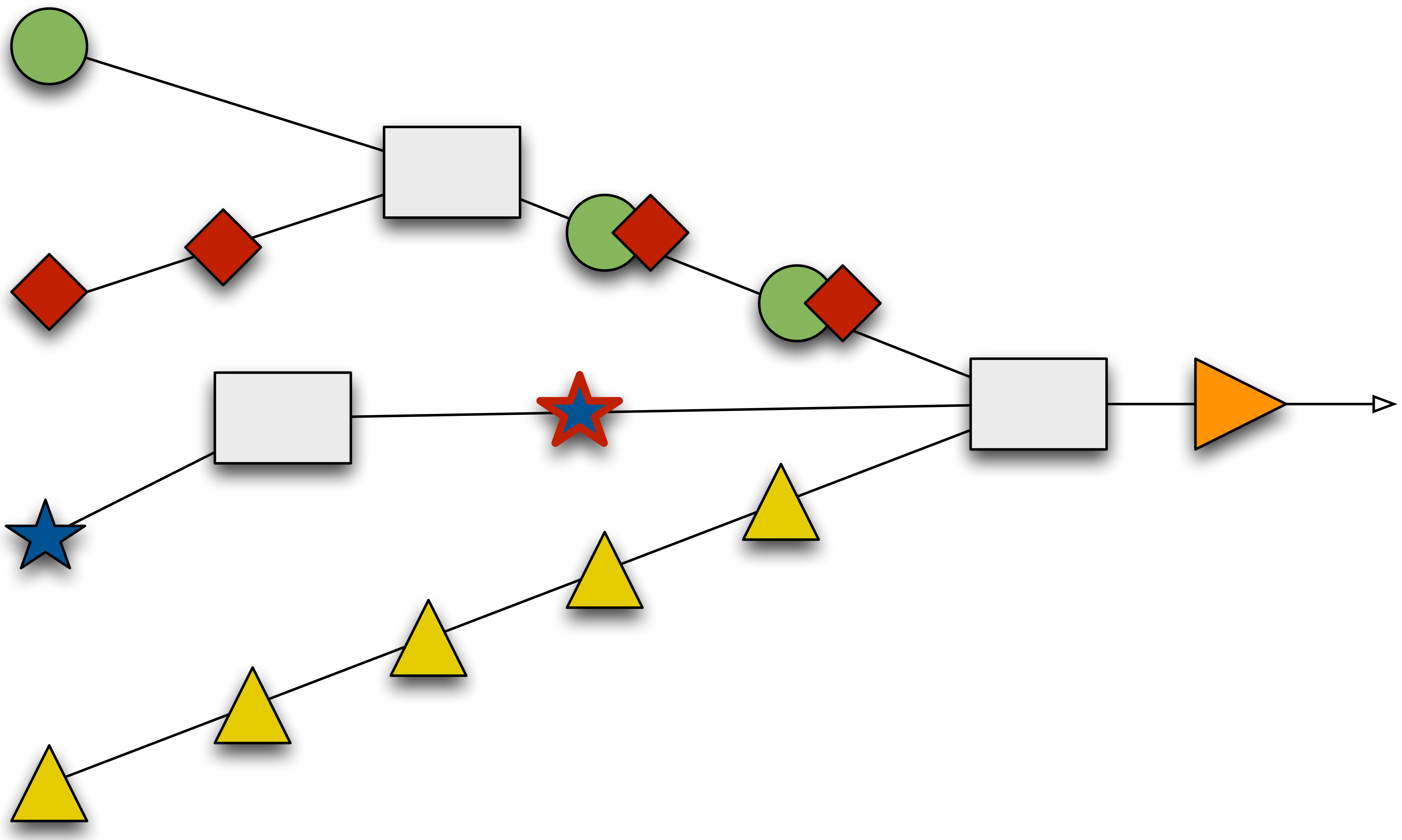
@benjchristensen

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

NETFLIX

http://techblog.netflix.com/

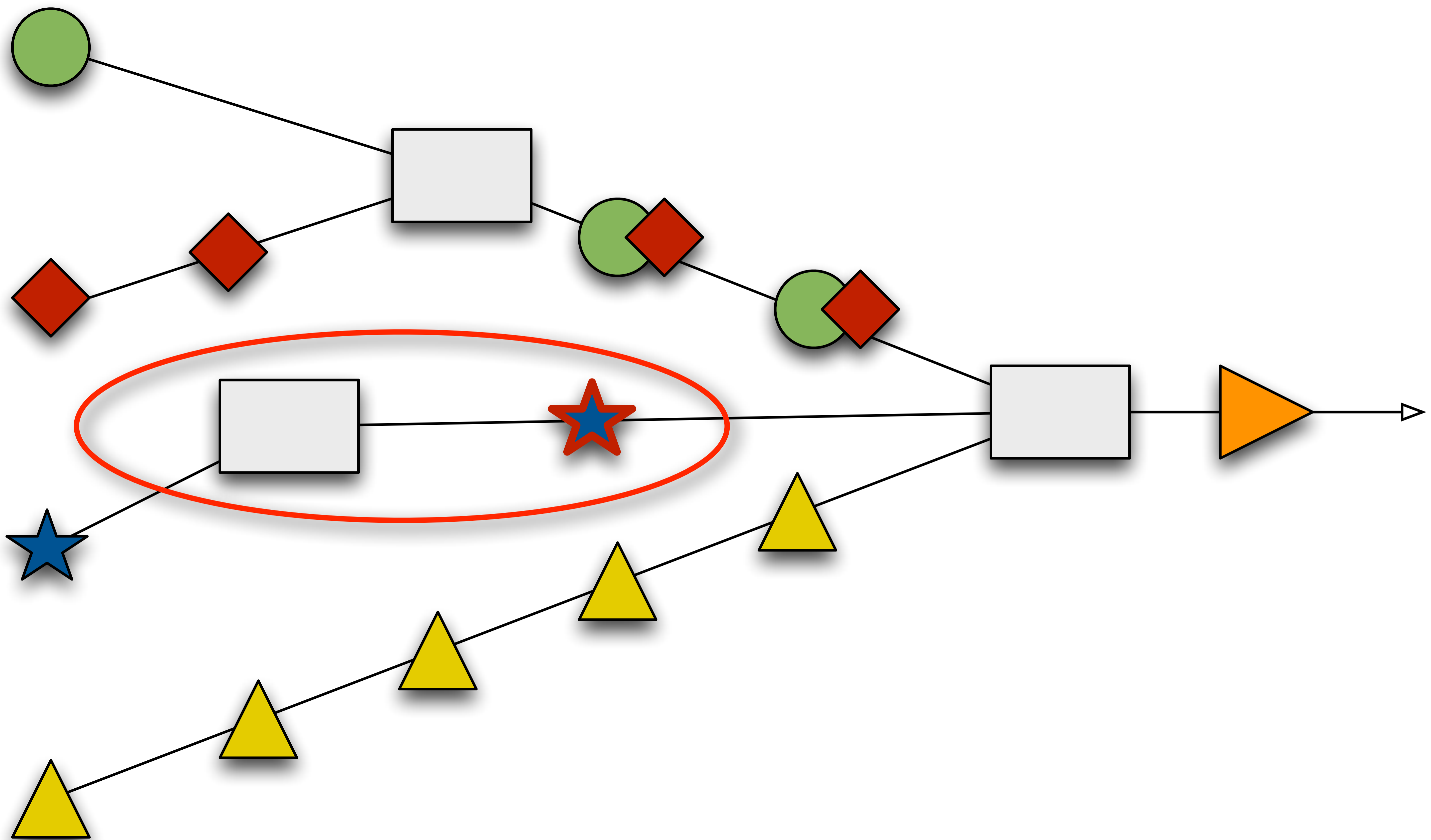
COMPOSABLE FUNCTIONS



REACTIVELY APPLIED

This presentation is about how the Netflix API application applies a functional programming style in an imperative Java application to apply functions reactively to asynchronously retrieved data ...

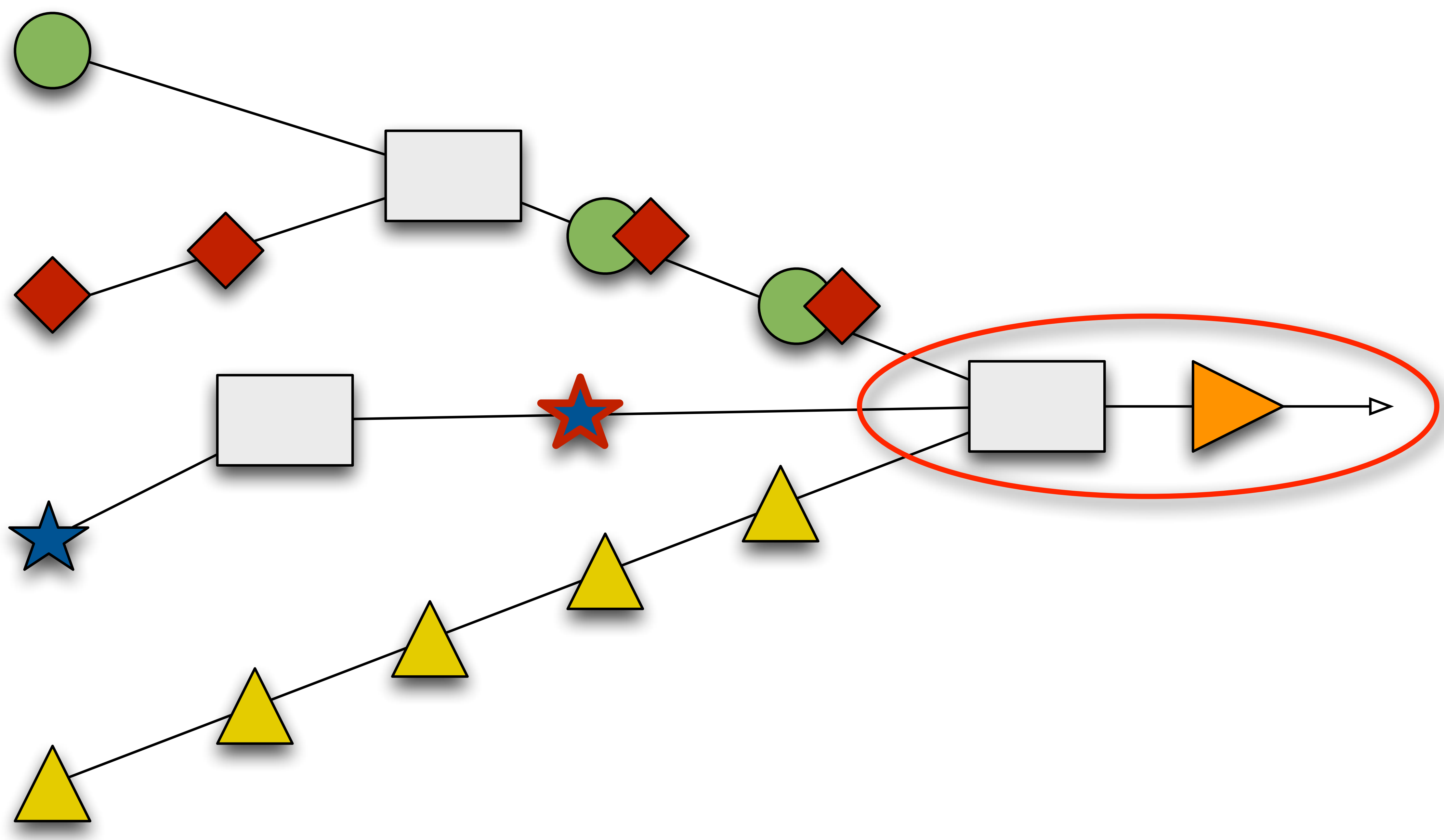
COMPOSABLE FUNCTIONS



REACTIVELY APPLIED

... and transform ...

COMPOSABLE FUNCTIONS



REACTIVELY APPLIED

... combine and output web service responses.

ASYNCHRONOUS

VALUES

EVENTS

PUSH

FUNCTIONAL REACTIVE

LAMBDAS

CLOSURES

(MOSTLY) PURE

COMPOSABLE

We have been calling this approach “functional reactive” since we use functions (lambdas/closures) in a reactive (asynchronous/push) manner.

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);
     });
```

Simple examples showing RxJava code in various languages supported by RxJava (https://github.com/Netflix/RxJava/tree/master/language-adaptors). Java8 works with rxjava-core and does not need a language-adaptor. It also works with Java 6/7 but without lambdas/closures the code is more verbose.

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);
      });
```

Most examples in the rest of this presentation will be in Groovy ...

Clojure

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

Scala

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

Groovy

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

JRuby

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

Java8

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

... with a handful in Java 8

RxJava

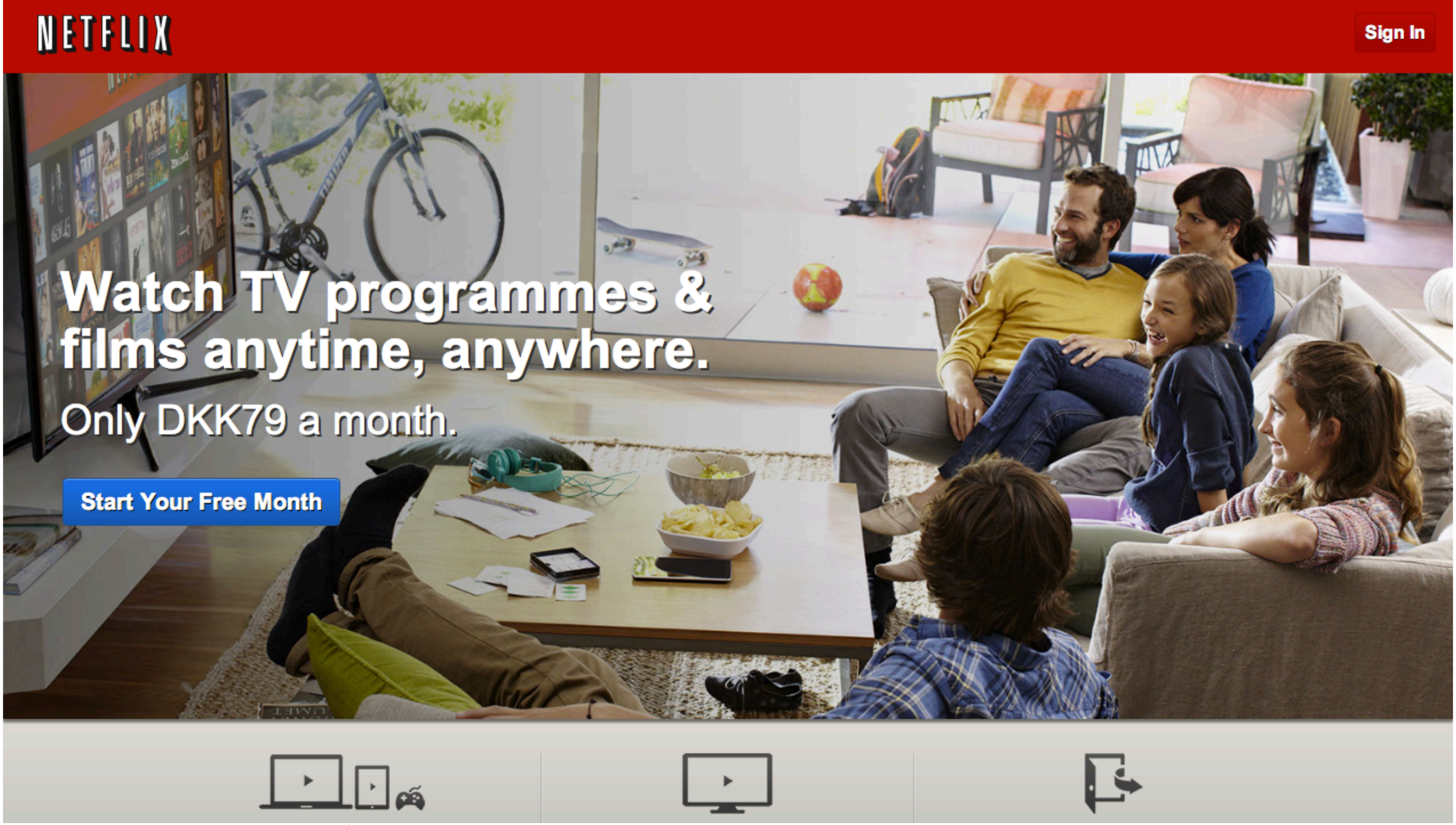
http://github.com/Netflix/RxJava

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

A Java port of Rx (Reactive Extensions)

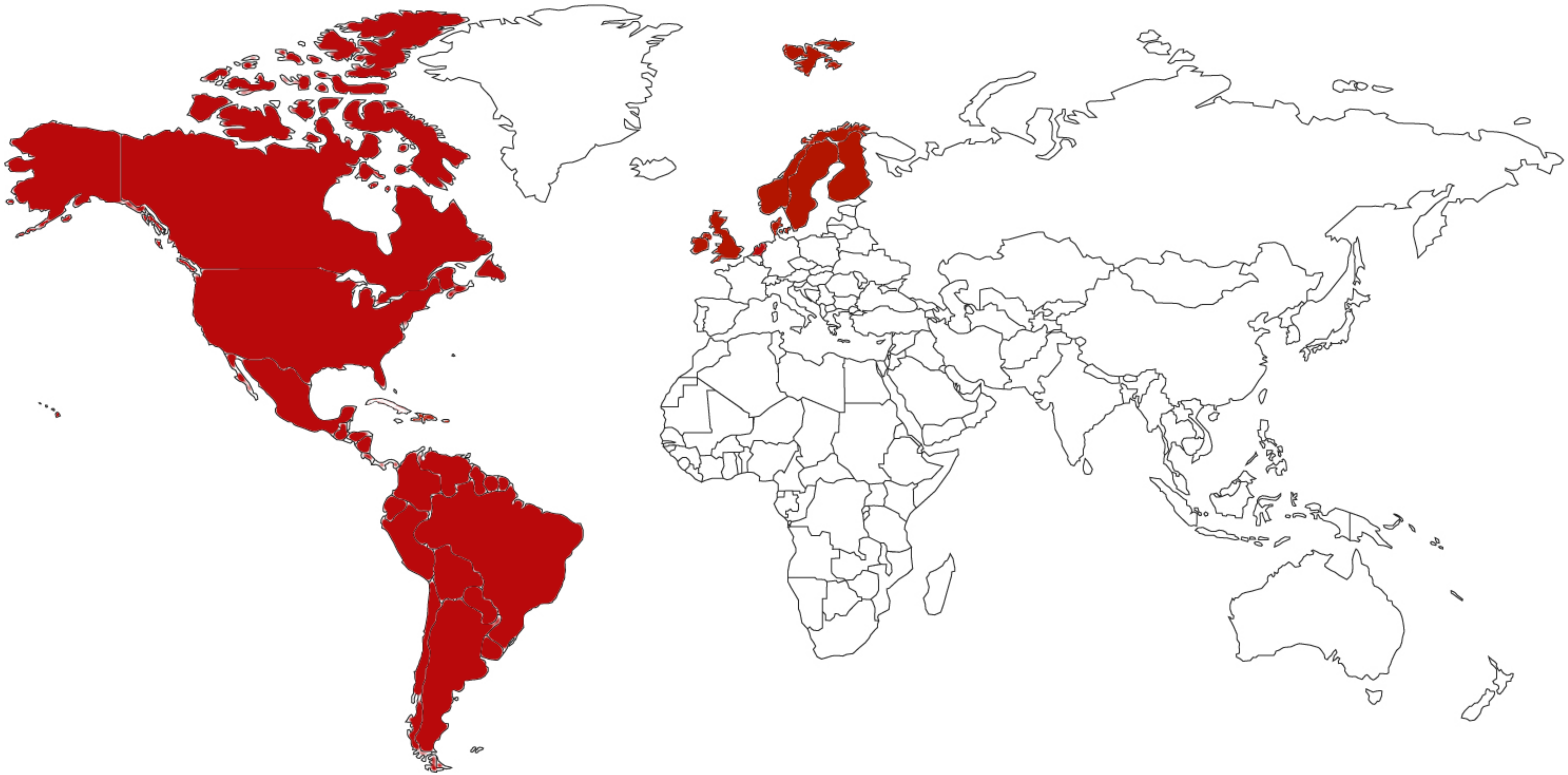
https://rx.codeplex.com (.Net and Javascript by Microsoft)

RxJava is a port of Microsoft's Rx (Reactive Extensions) to Java that attempts to be polyglot by targeting the JVM rather than just Java the language.



Netflix is a subscription service for movies and TV shows for $7.99USD/month (about the same converted price in each countries local currency).

More than 37 million Subscribers in 50+ Countries and Territories



Netflix has over 37 million video streaming customers in 50+ countries and territories across North & South America, United Kingdom, Ireland, Netherlands and the Nordics.

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

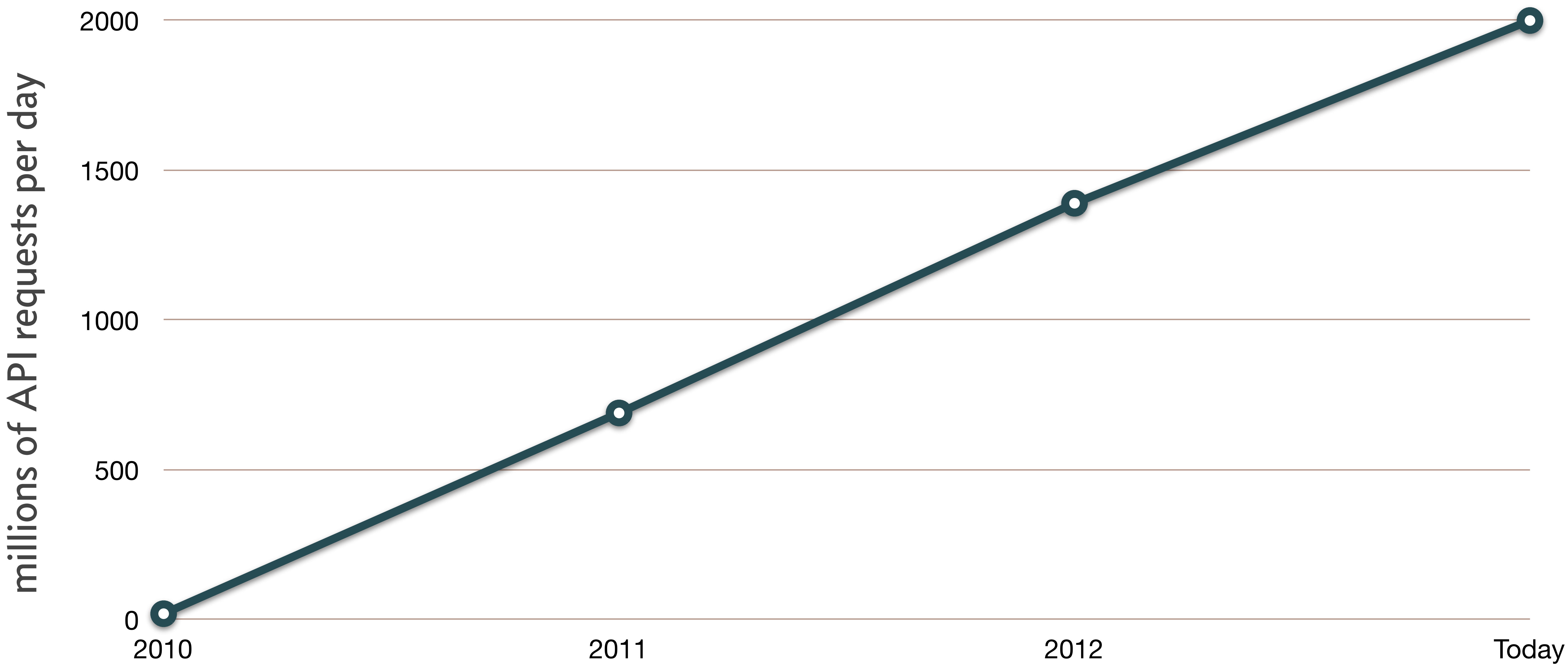
	Upstream		Downstream		Aggregate	
Rank	Application	Share	Application	Share	Application	Share
1	BitTorrent	36.8%	Netflix	33.0%	Netflix	28.8%
2	HTTP	9.83%	YouTube	14.8%	YouTube	13.1%
3	Skype	4.76%	HTTP	12.0%	HTTP	11.7%
4	Netflix	4.51%	BitTorrent	5.89%	BitTorrent	10.3%
5	SSL	3.73%	iTunes	3.92%	iTunes	3.43%
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%**

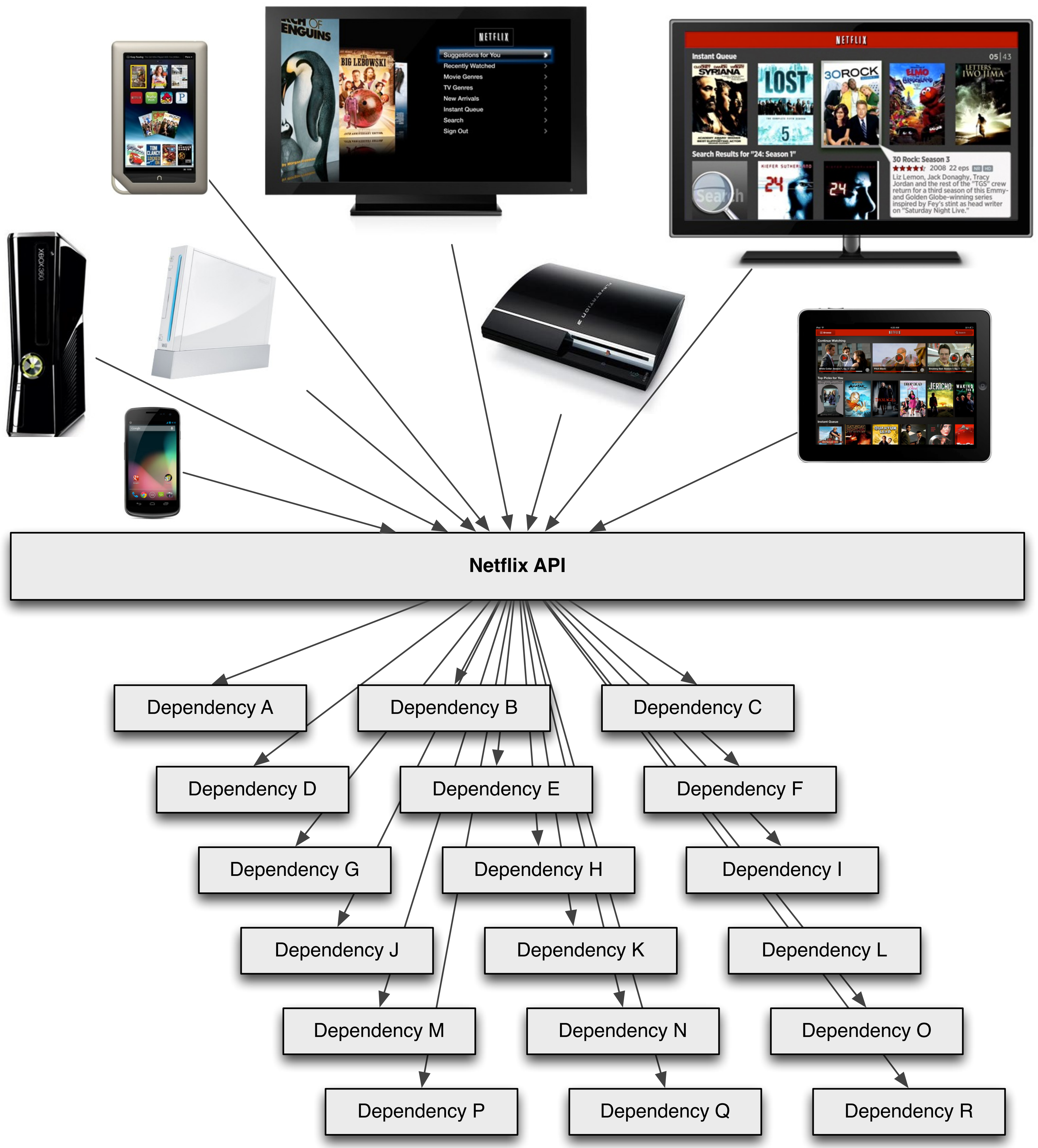
sandvine

Netflix subscribers are watching more than 1 billion hours a month

Sandvine report available with free account at http://www.sandvine.com/news/global_broadband_trends.asp

API traffic has grown from ~20 million/day in 2010 to >2 billion/day



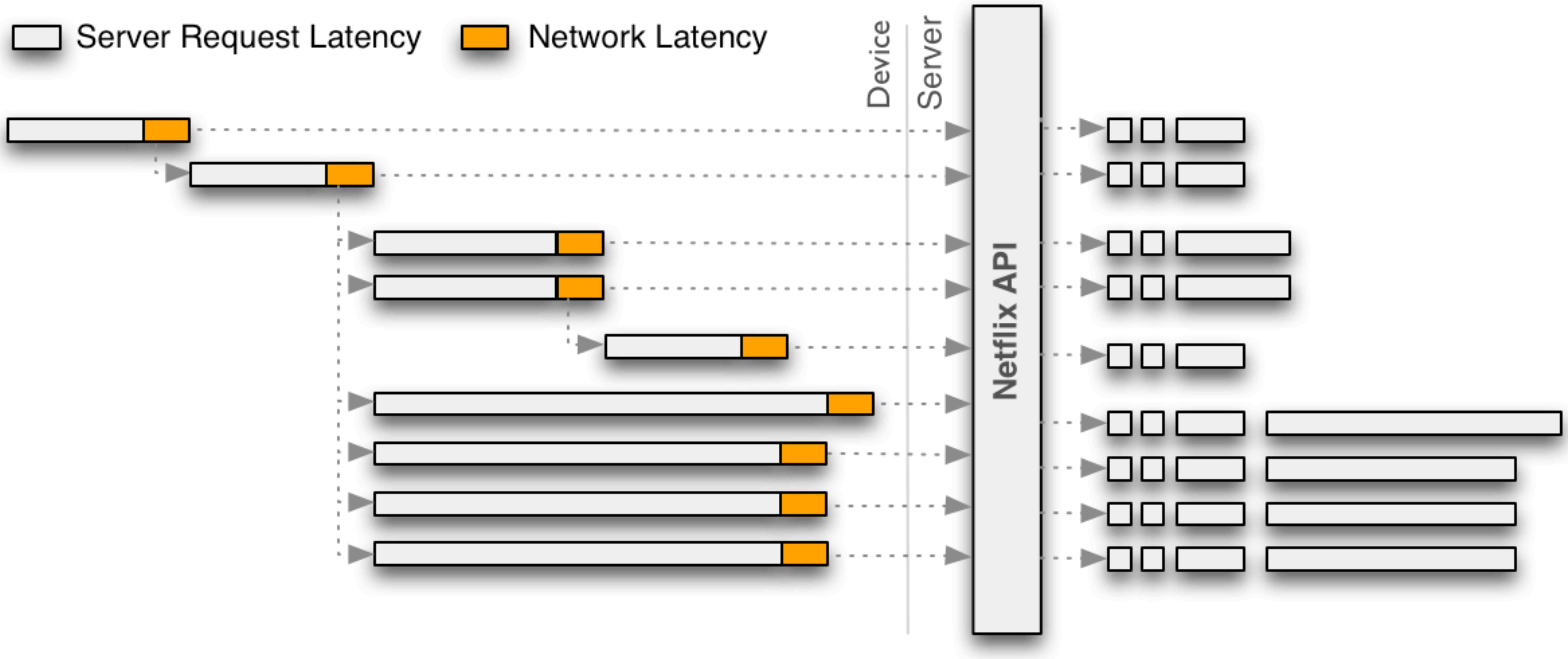


The Netflix API serves all streaming devices and acts as the broker between backend Netflix systems and the user interfaces running on the 1000+ devices that support Netflix streaming.

This presentation is going to focus on why the Netflix API team chose the functional reactive programming model (Rx in particular), how it is used and what benefits have been achieved.

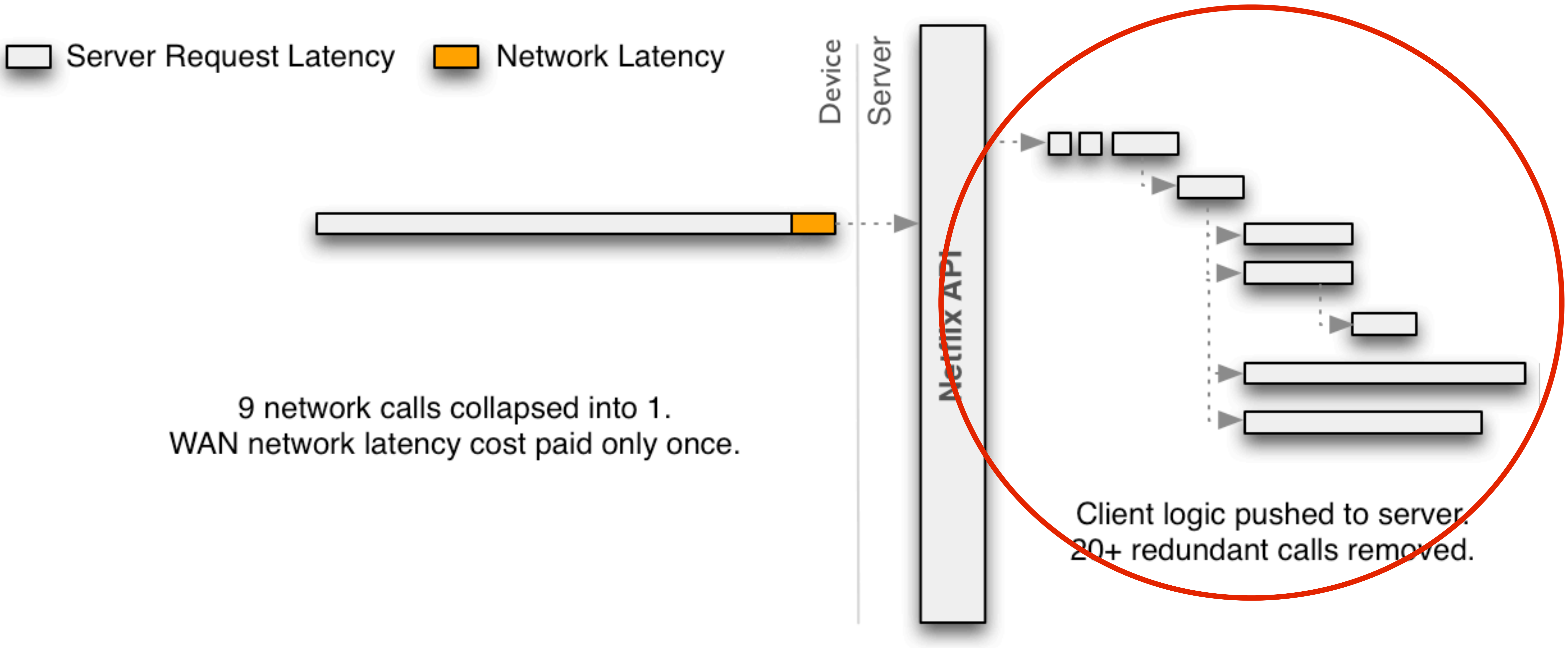
Other aspects of the Netflix API architecture can be found at http://techblog.netflix.com/search/label/api and https://speakerdeck.com/benjchristensen/.

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



More information about the re-architecture can be found at http://techblog.netflix.com/2013/01/optimizing-netflix-api.html

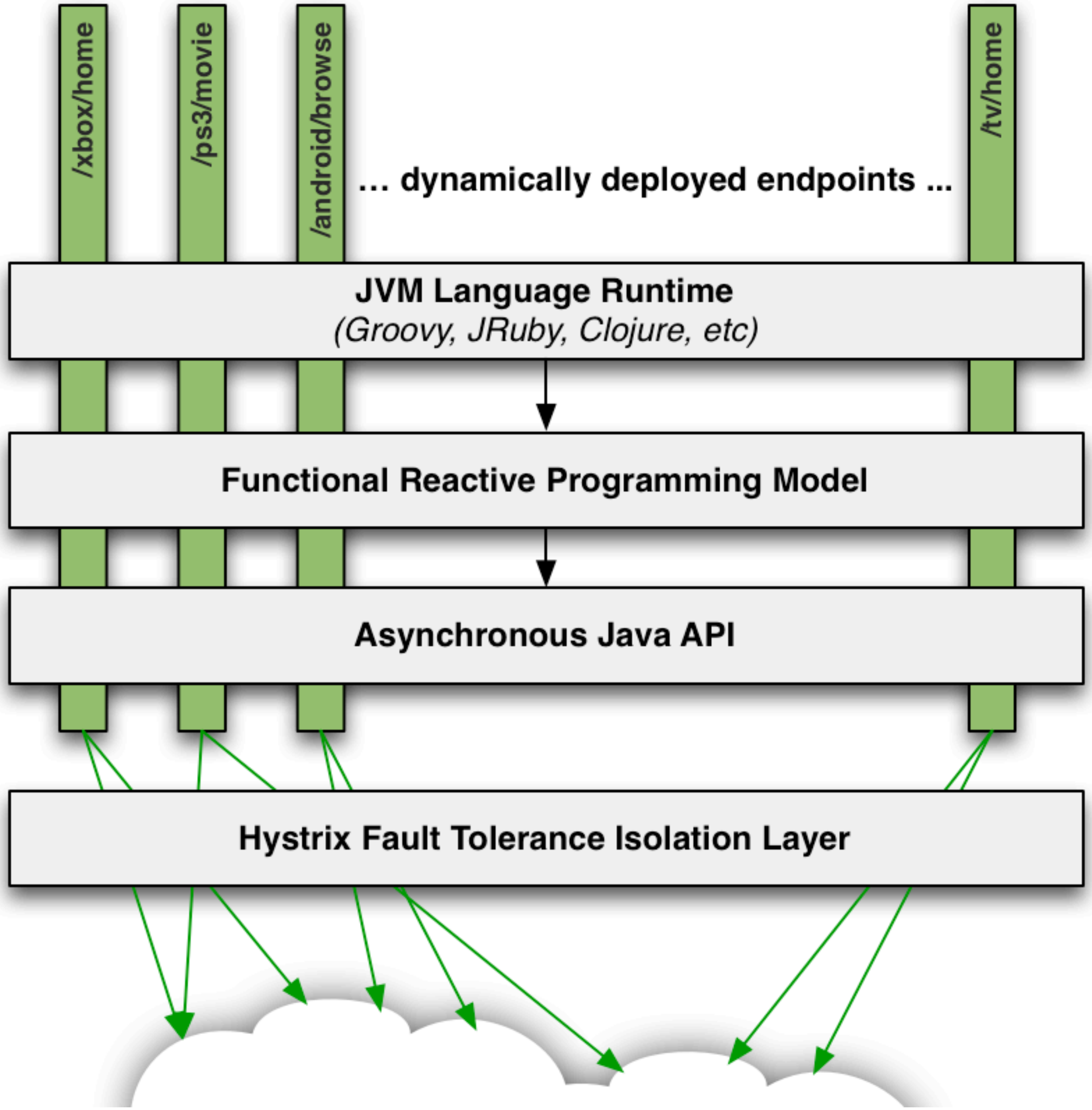
… that collapsed network traffic into coarse API calls …



NESTED, CONDITIONAL, CONCURRENT EXECUTION

Within a single request we now must achieve at least the same level of concurrency as previously achieved by the parallel network requests and preferably better as we can leverage the power of server hardware, lower latency network communication and eliminate redundant calls performed per incoming request.

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

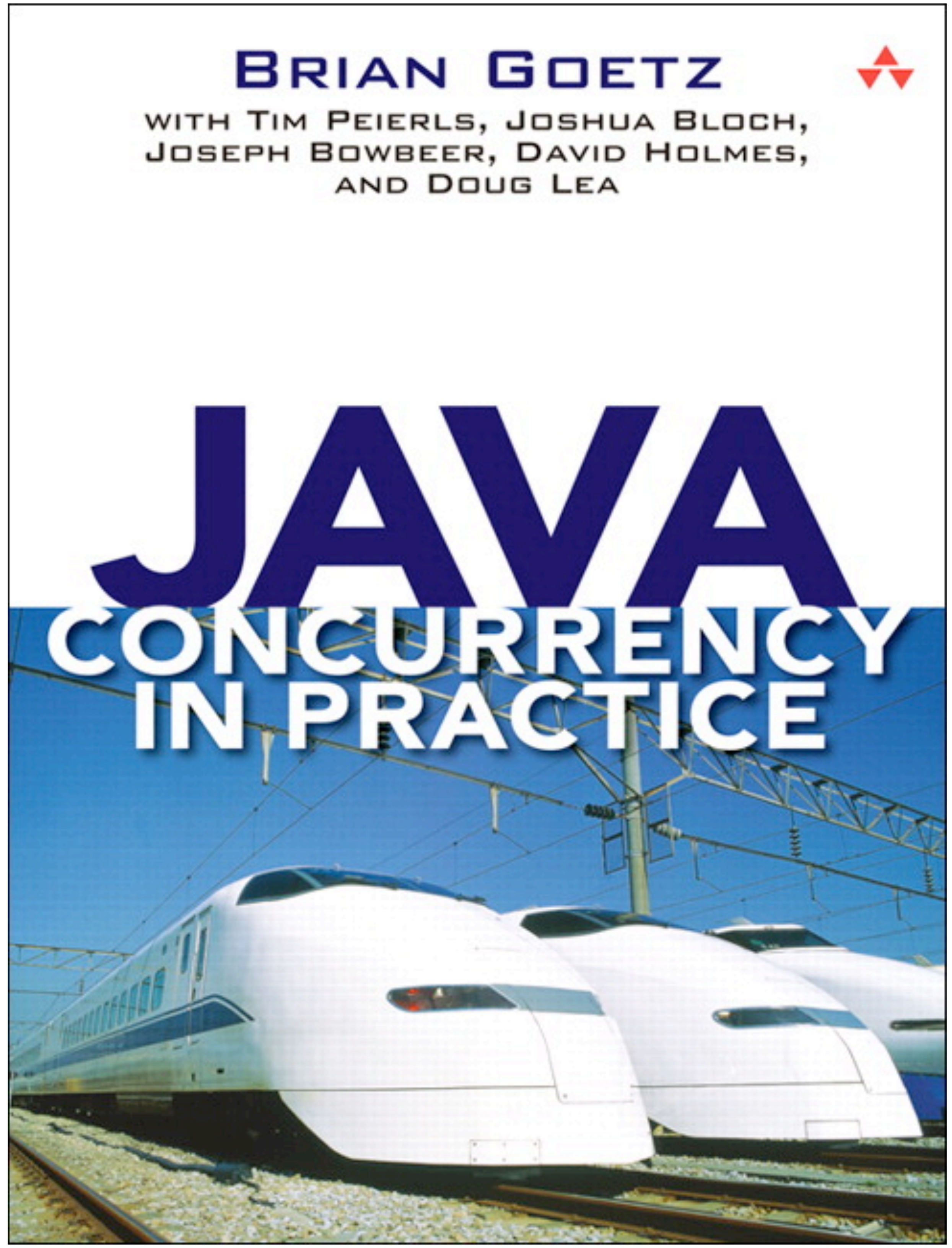


User interface client teams now build and deploy their own webservice endpoints on top of the API Platform instead of the "API Team" being the only ones who create endpoints.

We wanted to retain flexibility to use whatever JVM language we wanted as well as cater to the differing skills and backgrounds of engineers on different teams.

Groovy was the first alternate language we deployed in production on top of Java.

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?

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

```java
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()

Observable/Observer is the asynchronous dual to the synchronous Iterable/Iterator.

More information about the duality of Iterable and Observable can be found at http://csl.stanford.edu/~christos/pldi2010.fit/meijer.duality.pdf and http://codebetter.com/matthewpodwysocki/2009/11/03/introduction-to-the-reactive-framework-part-ii/

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})
```

The same way higher-order functions can be applied to an Iterable they can be applied to an Observable.

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()

Grid of synchronous and asynchronous duals for single and multi-valued responses. The Rx Observable is the dual of the synchronous Iterable.

	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
}
```

Typical synchronous scalar response with subsequent conditional logic.

	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
    }
}
```

Similar to scalar value except conditional logic happens within a loop.

	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
}
```

As we move to async a normal Java Future is asynchronous but to apply conditional logic requires dereferencing the value via ‘get()’.

	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
}
```

And this leads to the typical issue in nested, conditional asynchronous code with Java Futures where asynchronous quickly becomes synchronous and blocking again.

	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);
```

There are better Futures though, one of them is from Guava ...

	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);
```

... and it allows callbacks ...

	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);
```

... so the conditional logic can be put inside a callback and prevent us from blocking and we can chain calls together in these callbacks.

	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
     }
});
```

New CompletableFuture in Java 8 are composable with higher-order functions.

	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
    }
});
```

Akka/Scala Futures are also composable and provide higher-order functions ...

	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
    }
});
```

... that get us to where we want to be so that we can now compose conditional, nested data flows while remaining asynchronous.

	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
    }
});
```

The composable Future ...

	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
    }
});
```

... is very similar to the Rx Observable except that an Rx Observable supports multiple values which means it can handle a single value, a sequence of values or an infinite stream.

	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
    }
});
```

We wanted to be asynchronous to abstract away the underlying concurrency decisions and composable Futures or Rx Observables are good solutions.

	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
    }
});
```

One reason we chose the Rx Observable is because it gives us a single abstraction that accommodates our needs for both single and multi-valued responses while giving us the higher-order functions to compose nested, conditional logic in a reactive manner.

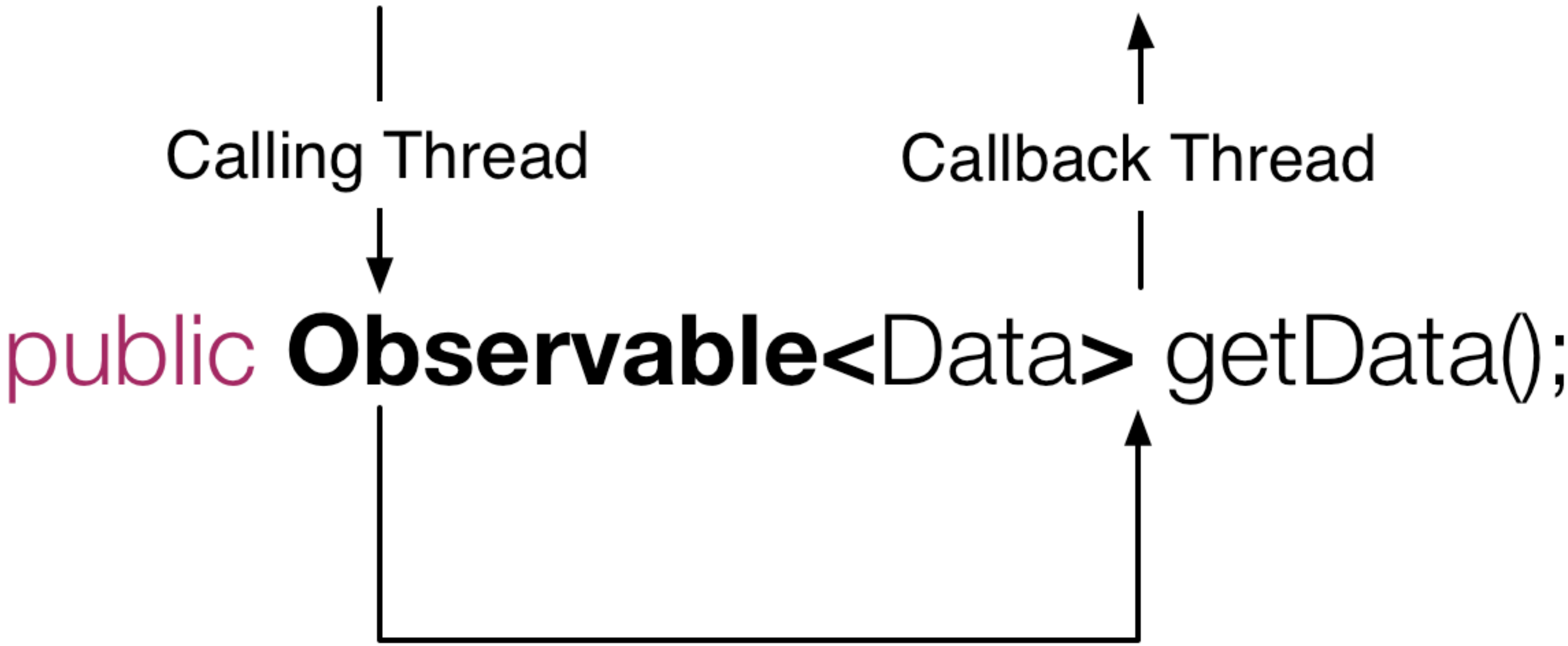
INSTEAD OF A BLOCKING API ...

```
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);
}
```

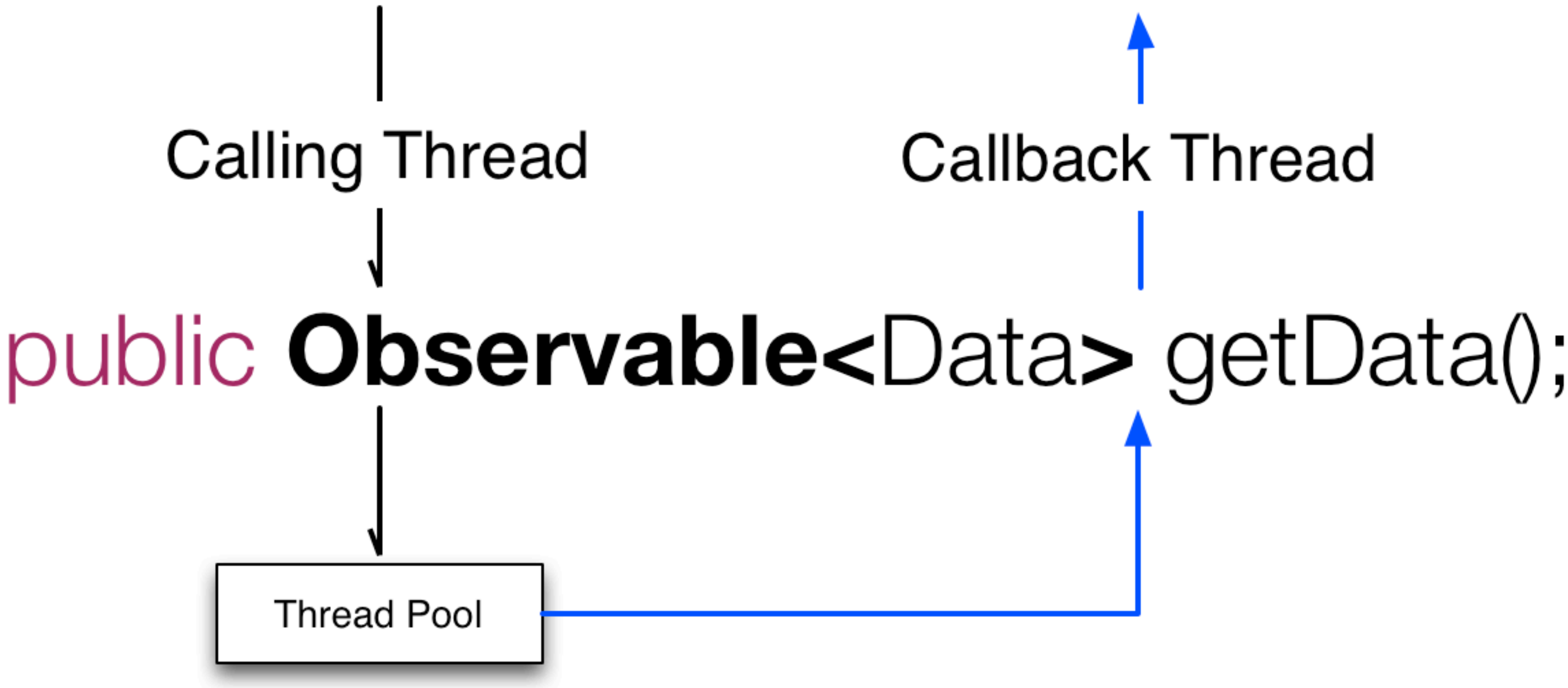
With Rx blocking APIs could be converted into Observable APIs and accomplish our architecture goals including abstracting away the control and implementation of concurrency and asynchronous execution.



Do work synchronously on calling thread.

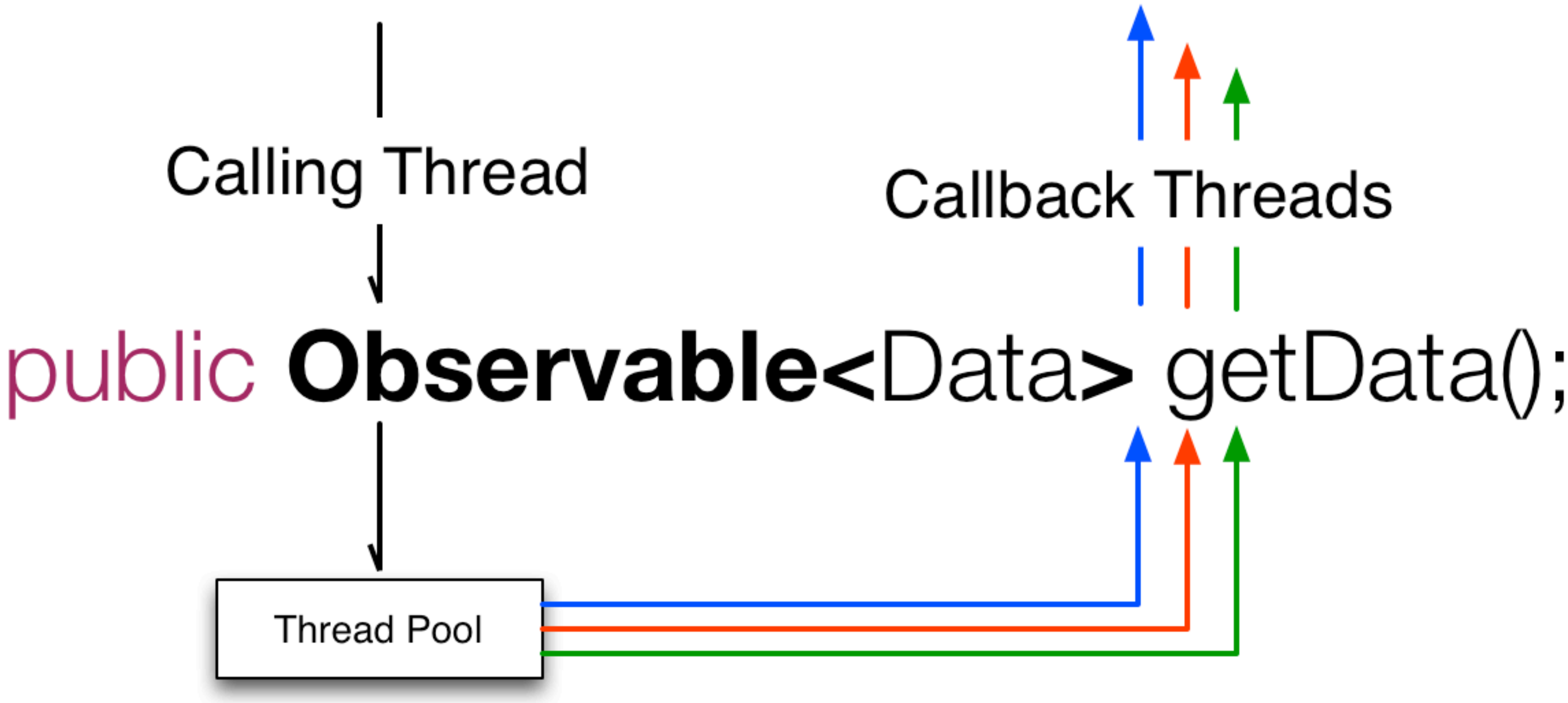
One of the other positives of Rx Observable was that it is abstracted from the source of concurrency. It is not opinionated and allows the implementation to decide.

For example, an Observable API could just use the calling thread to synchronously execute and respond.



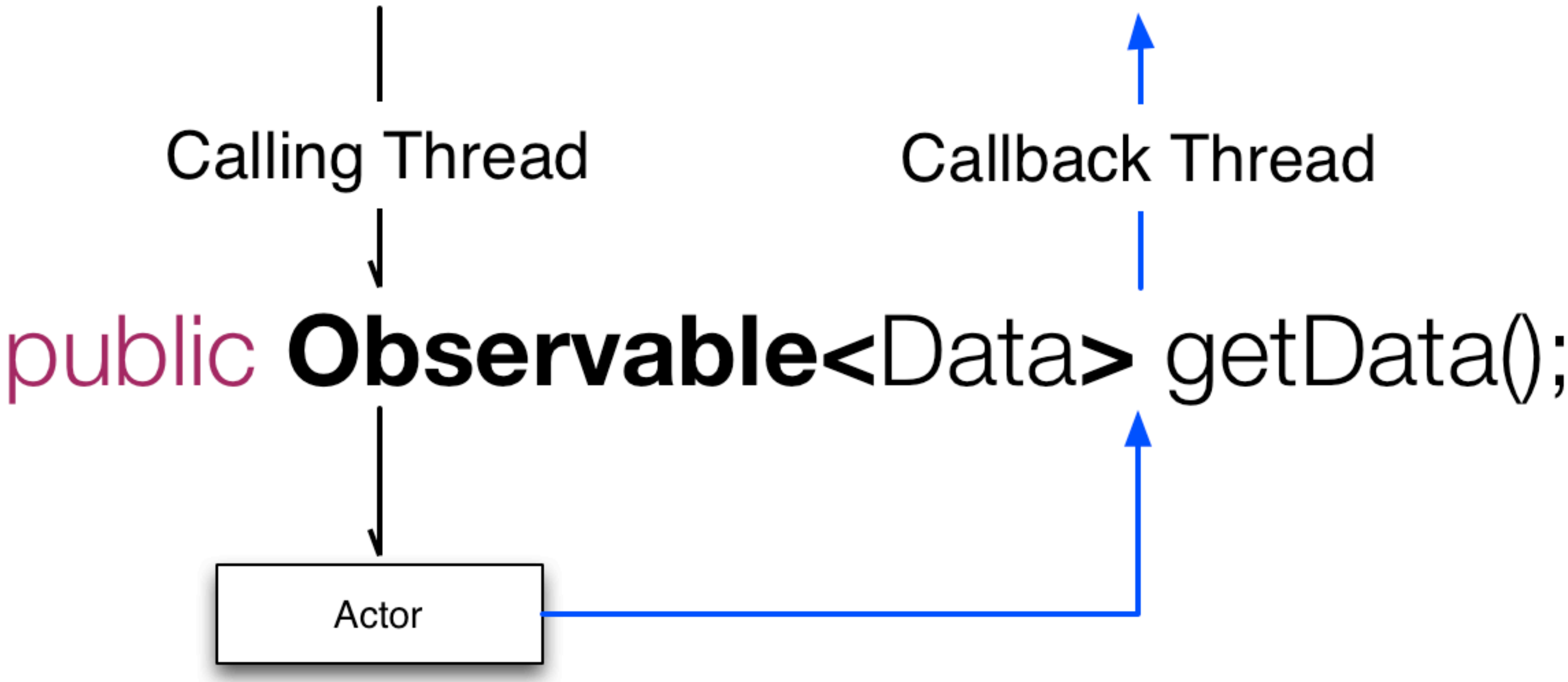
Do work asynchronously on a separate thread.

Or it could use a thread-pool to do the work asynchronously and callback with that thread.



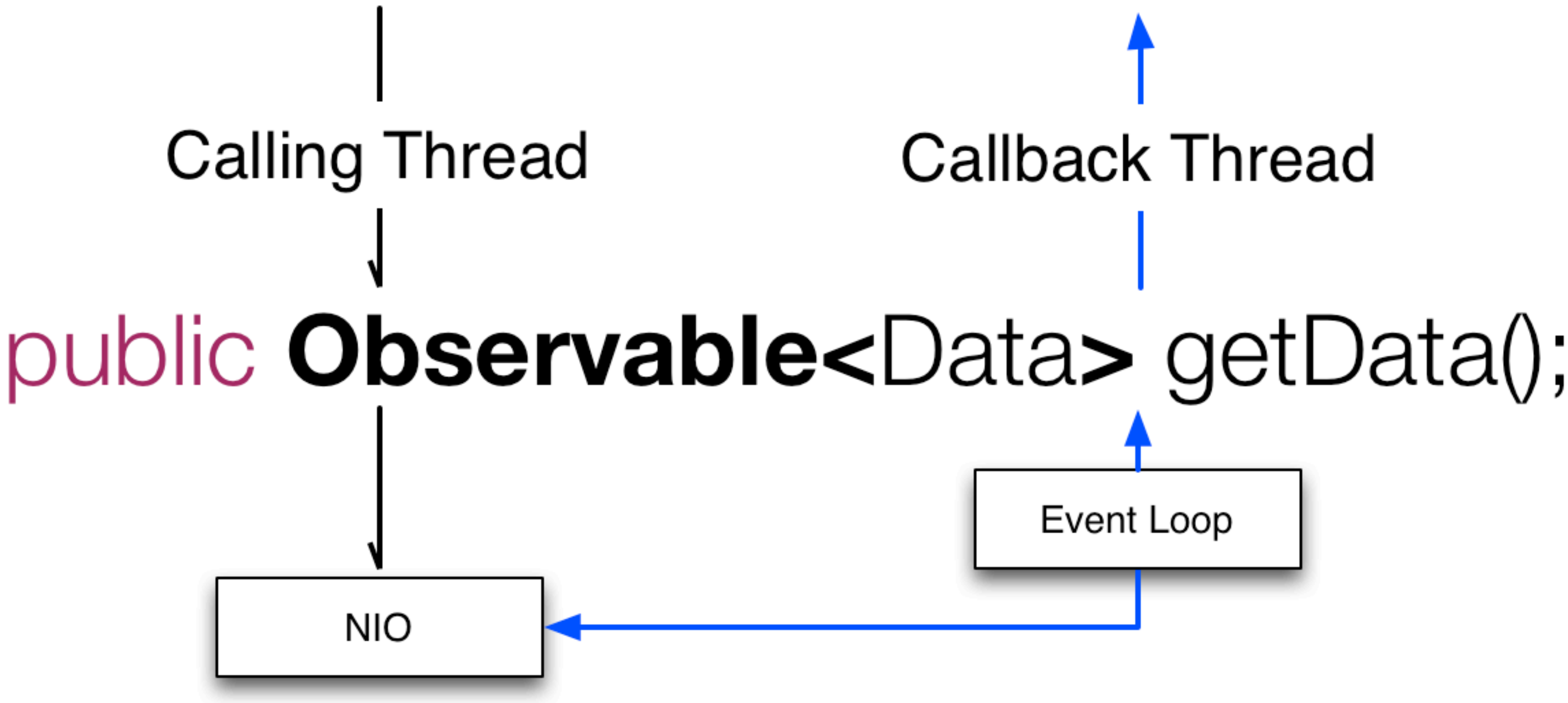
Do work asynchronously on a multiple threads.

Or it could use multiple threads, each thread calling back via onNext(T) when the value is ready.



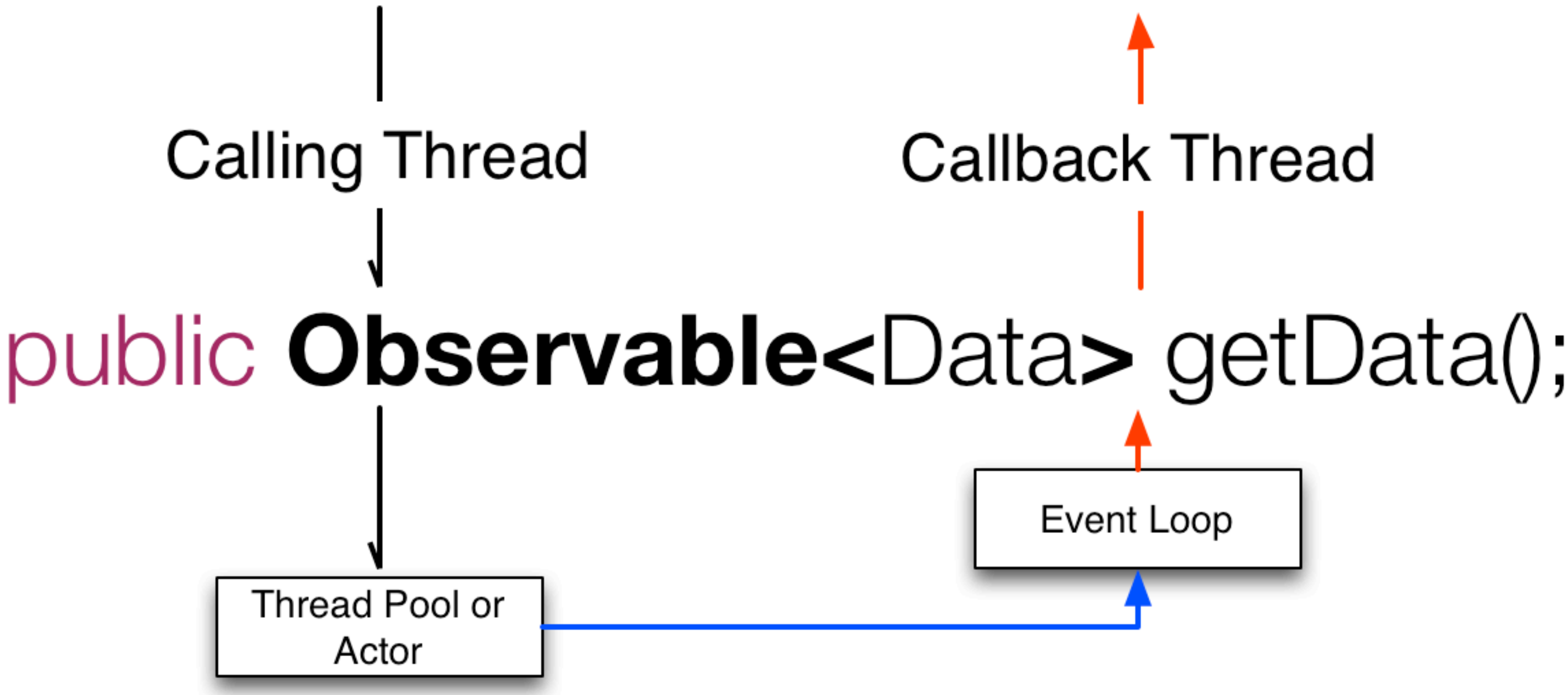
Do work asynchronously on an actor
(or multiple actors).

Or it could use an actor pattern instead of a thread-pool.



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

Or NIO with an event-loop.



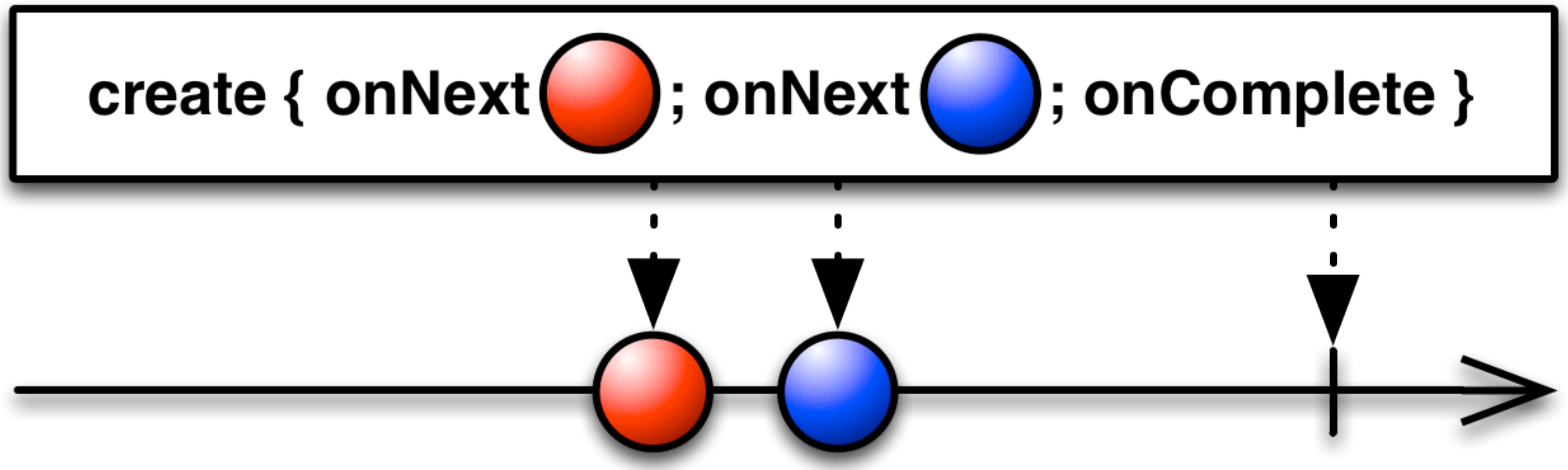
Do work asynchronously and perform callback via a single or multi-threaded event loop.

Or a thread-pool/actor that does the work but then performs the callback via an event-loop so the thread-pool/actor is tuned for IO and event-loop for CPU.

All of these different implementation choices are possible without changing the signature of the method and without the calling code changing their behavior or how they interact with or compose responses.

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

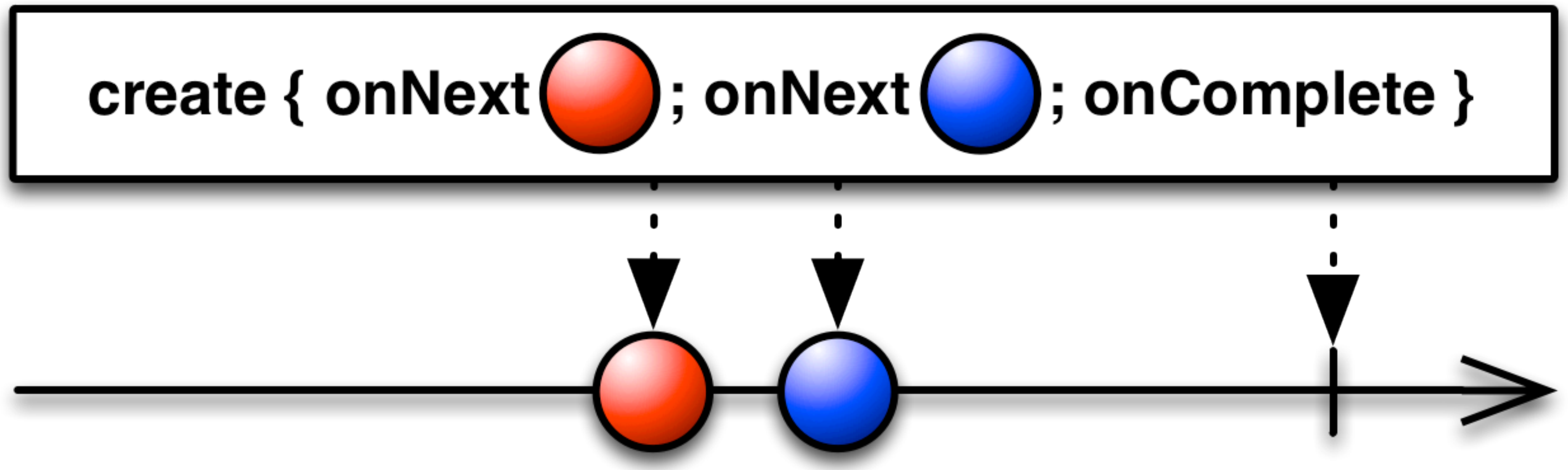



```
Observable<T> create(Func1<Observer<T>, Subscription> func)
```

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

Let's look at how to create an Observable and what its contract is. An Observable receives an Observer and calls onNext 1 or more times and terminates by either calling onError or onCompleted once.

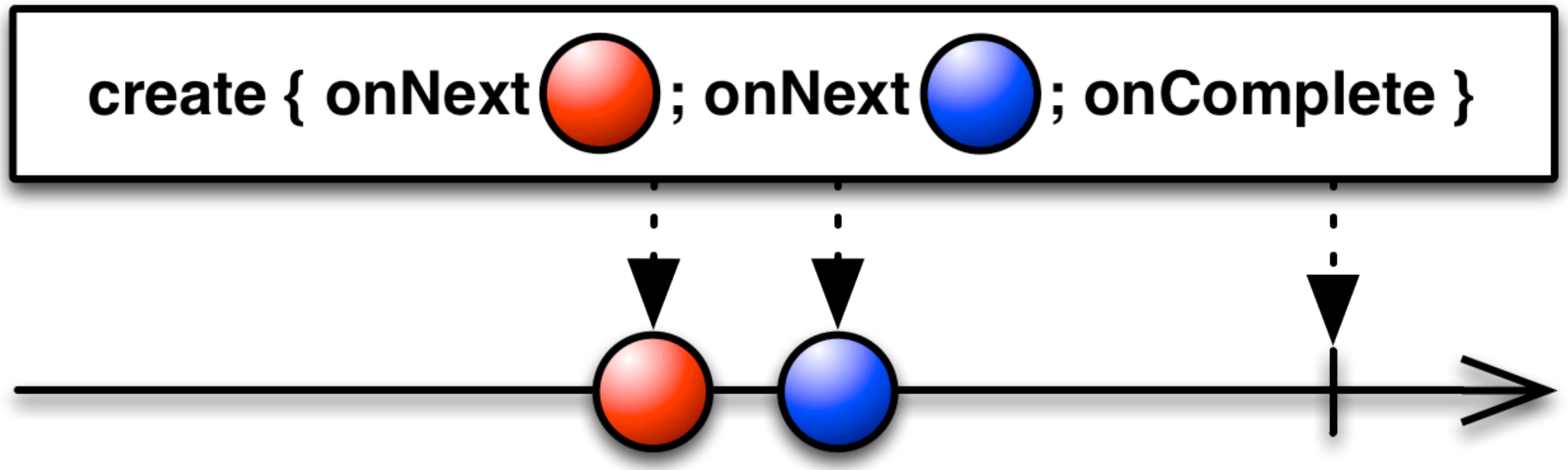
More information is available at https://github.com/Netflix/RxJava/wiki/Observable




```
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);
  }
})
```

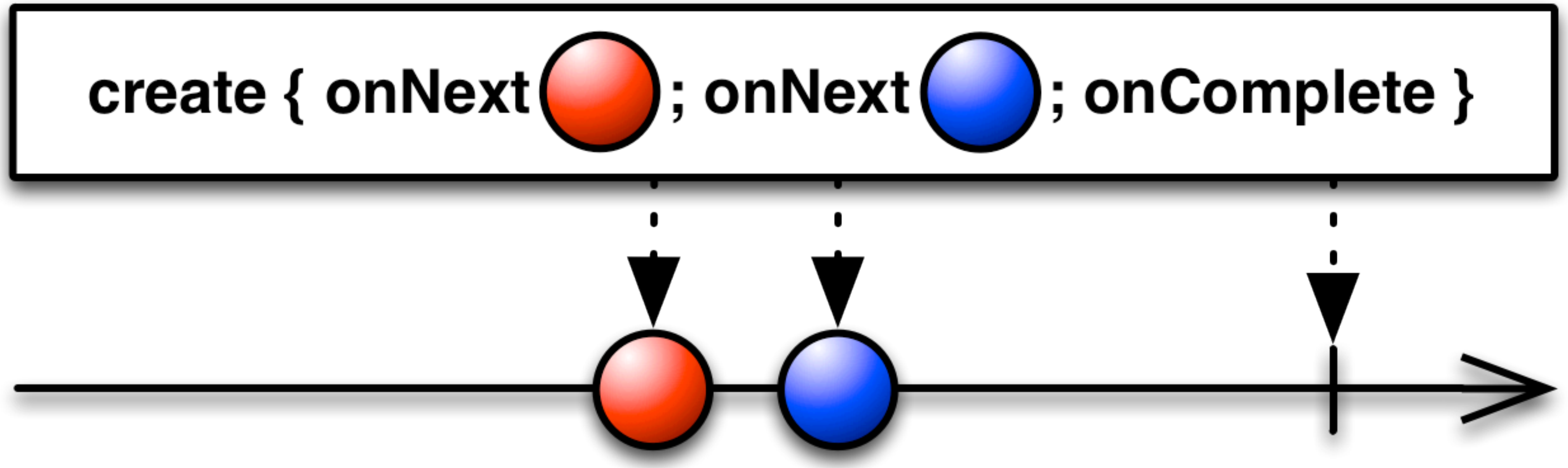
An Observable is created by passing a Func1 implementation...




```
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);
  }
})
```

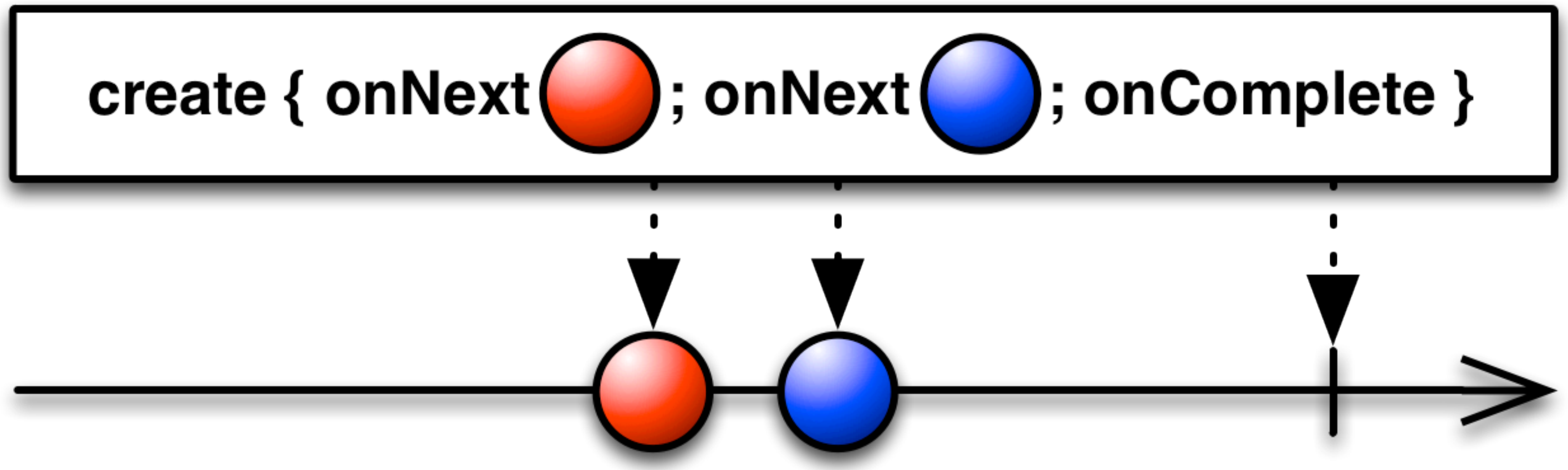
... that accepts an Observer ...




```
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);
  }
})
```

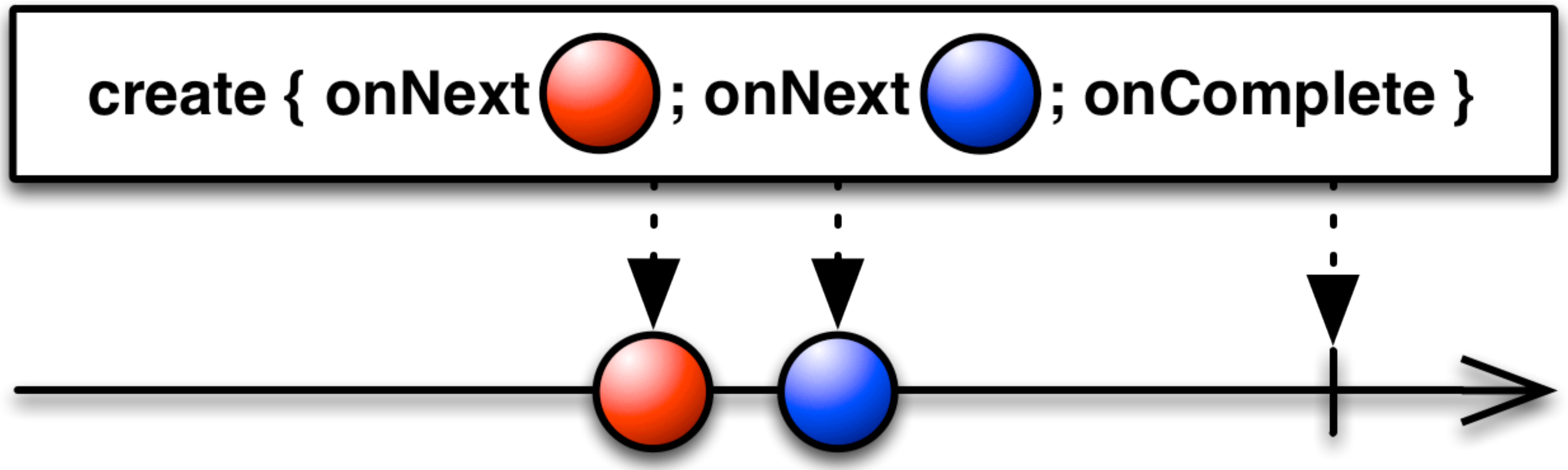
... and when executed (subscribed to) it emits data via ‘onNext’ ...




```
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);
  }
})
```

... and marks its terminal state by calling ‘onCompleted’ ...




```
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);
  }
})
```

... or ‘onError’ if a failure occurs. Either ‘onCompleted’ or ‘onError’ must be called to terminate an Observable and nothing can be called after the terminal state occurs. An infinite stream that never has a failure would never call either of these.

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);
               }
            }
        })
    })
}
```

Example Observable implementation that executes asynchronously on a thread-pool and emits a single value. This explicitly shows an 'executor' being used to run this on a separate thread to illustrate how it is up to the Observable implementation to do as it wishes, but Rx always has Schedulers for typical scenarios of scheduling an Observable in a thread-pool or whatever a Scheduler implementation dictates.

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);
               }
            }
        })
    })
}
```

Example Observable implementation that executes asynchronously on a thread-pool and emits a single value. This explicitly shows an ‘executor’ being used to run this on a separate thread to illustrate how it is up to the Observable implementation to do as it wishes, but Rx always has Schedulers for typical scenarios of scheduling an Observable in a thread-pool or whatever a Scheduler implementation dictates.

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)

Example Observable implementation that executes synchronously and emits multiple values.

Note that the for-loop as implemented here will always complete so should not have any IO in it and be of limited length otherwise it should be done with a lazy iterator implementation or performed asynchronously so it can be unsubscribed from.

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)

Example Observable implementation that executes synchronously and emits multiple values.

Note that the for-loop as implemented here will always complete so should not have any IO in it and be of limited length otherwise it should be done with a lazy iterator implementation or performed asynchronously so it can be unsubscribed from.

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);
            }
         }
      })
   })
}
```

Example Observable implementation that executes asynchronously on a thread-pool and emits multiple values.

Note that for brevity this code does not handle the subscription so will not unsubscribe even if asked.

See the 'getListOfLists' method in the following for an implementation with unsubscribe handled: https://github.com/Netflix/RxJava/blob/master/language-adaptors/rxjava-groovy/src/examples/groovy/rx/lang/groovy/examples/VideoExample.groovy#L125

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);
                }
            }
        })
    })
}
```

Example Observable implementation that executes asynchronously on a thread-pool and emits multiple values.

Note that for brevity this code does not handle the subscription so will not unsubscribe even if asked.

See the 'getListOfLists' method in the following for an implementation with unsubscribe handled: https://github.com/Netflix/RxJava/blob/master/language-adaptors/rxjava-groovy/src/examples/groovy/rx/lang/groovy/examples/VideoExample.groovy#L125

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")
    }
})
```

Moving to the subscriber side of the relationship we see how an Observer looks. This implements the full interface for clarity of what the types and members are ...

Asynchronous Observer

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

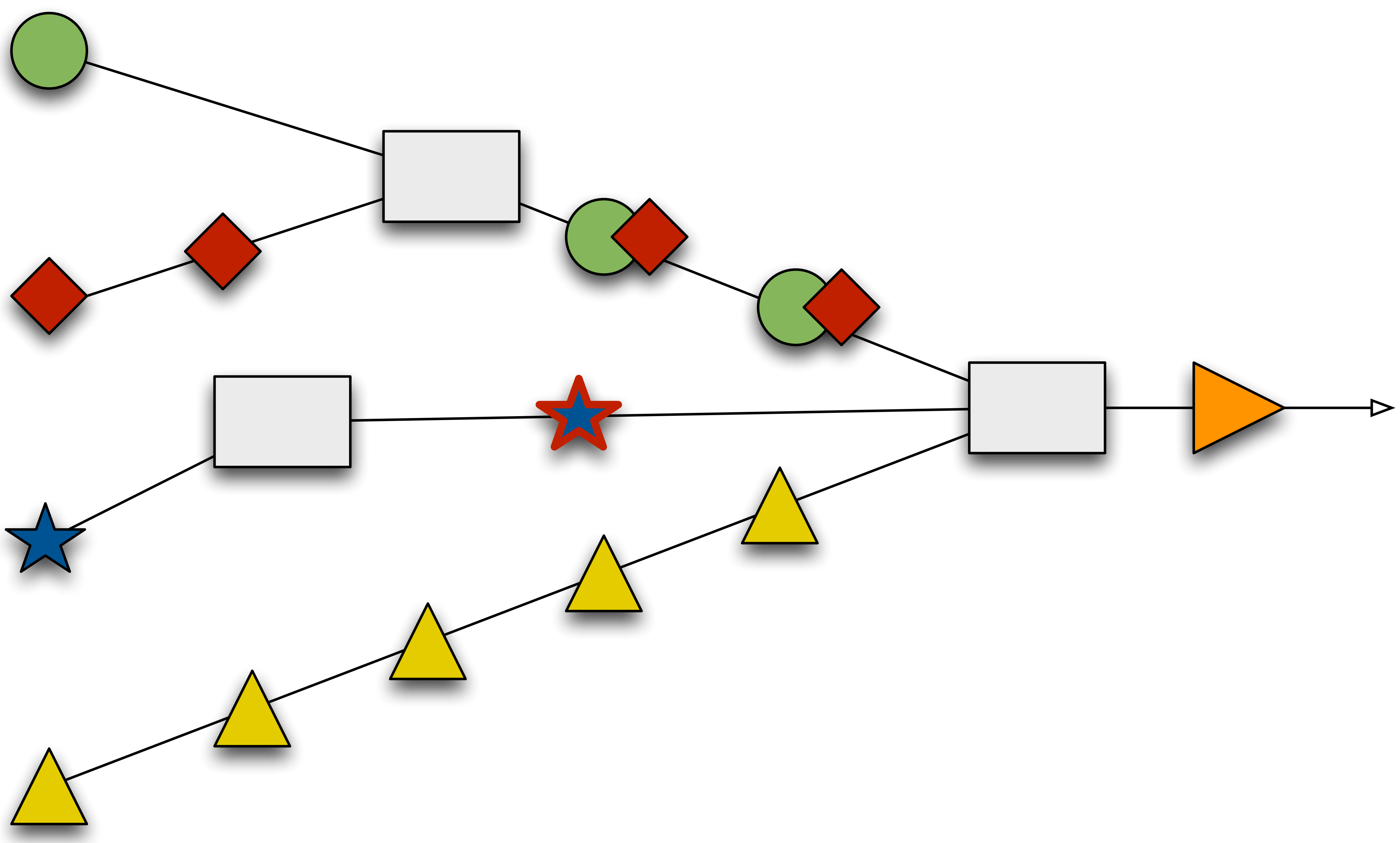
... but generally the on* method implementations are passed in as functions/lambdas/closures similar to this.

Asynchronous Observer

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

Often the ‘onCompleted’ function is not needed.

COMPOSABLE FUNCTIONS



The real power though is when we start composing Observables together.

COMPOSABLE FUNCTIONS

Transform: Map, FlatMap, Reduce, Scan ...

Filter: Take, Skip, Sample, TakeWhile, Filter ...

Combine: Concat, Merge, Zip, CombineLatest,
Multicast, Publish, Cache, RefCount ...

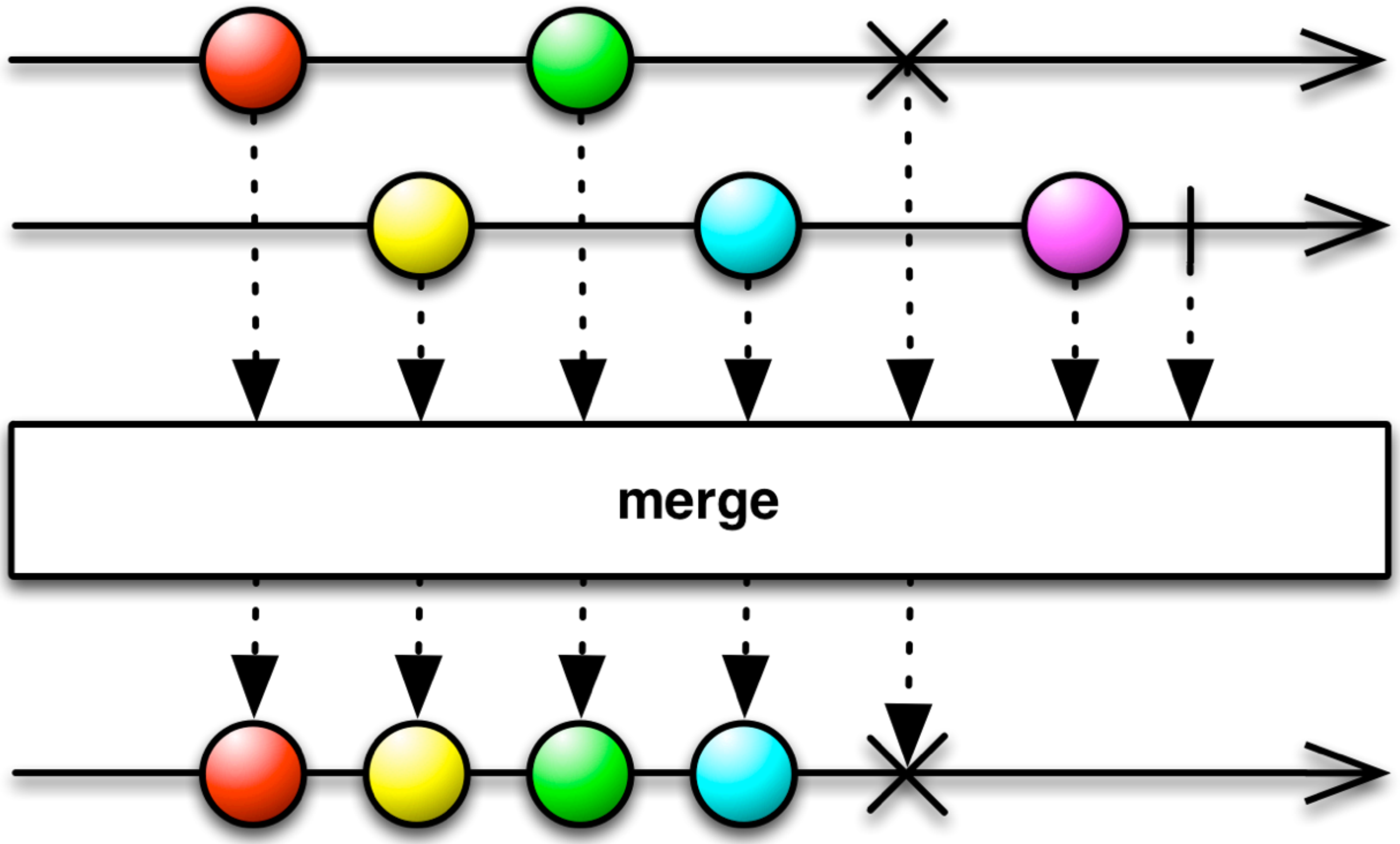
Concurrency: ObserveOn, SubscribeOn

Error Handling: OnErrorReturn, OnErrorResume ...

This is a list of some of the higher-order functions that Rx supports. More can be found in the documentation (https://github.com/Netflix/RxJava/wiki) and many more from the original Rx.Net implementation have not yet been implemented in RxJava (but are all listed on the RxJava Github issues page tracking the progress).

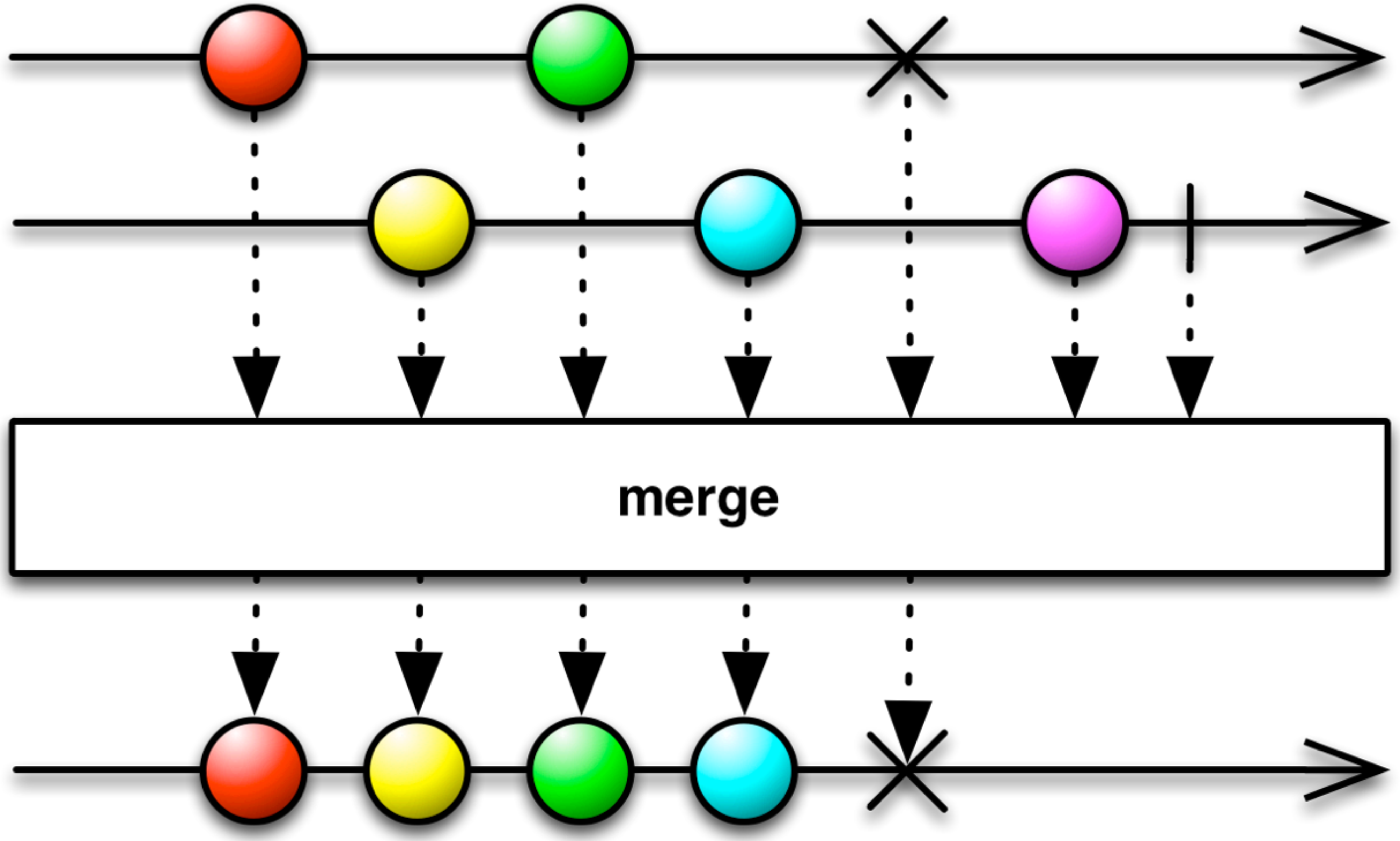
We will look at some of the important ones for combining and transforming data as well as handling errors asynchronously.

Combining via Merge

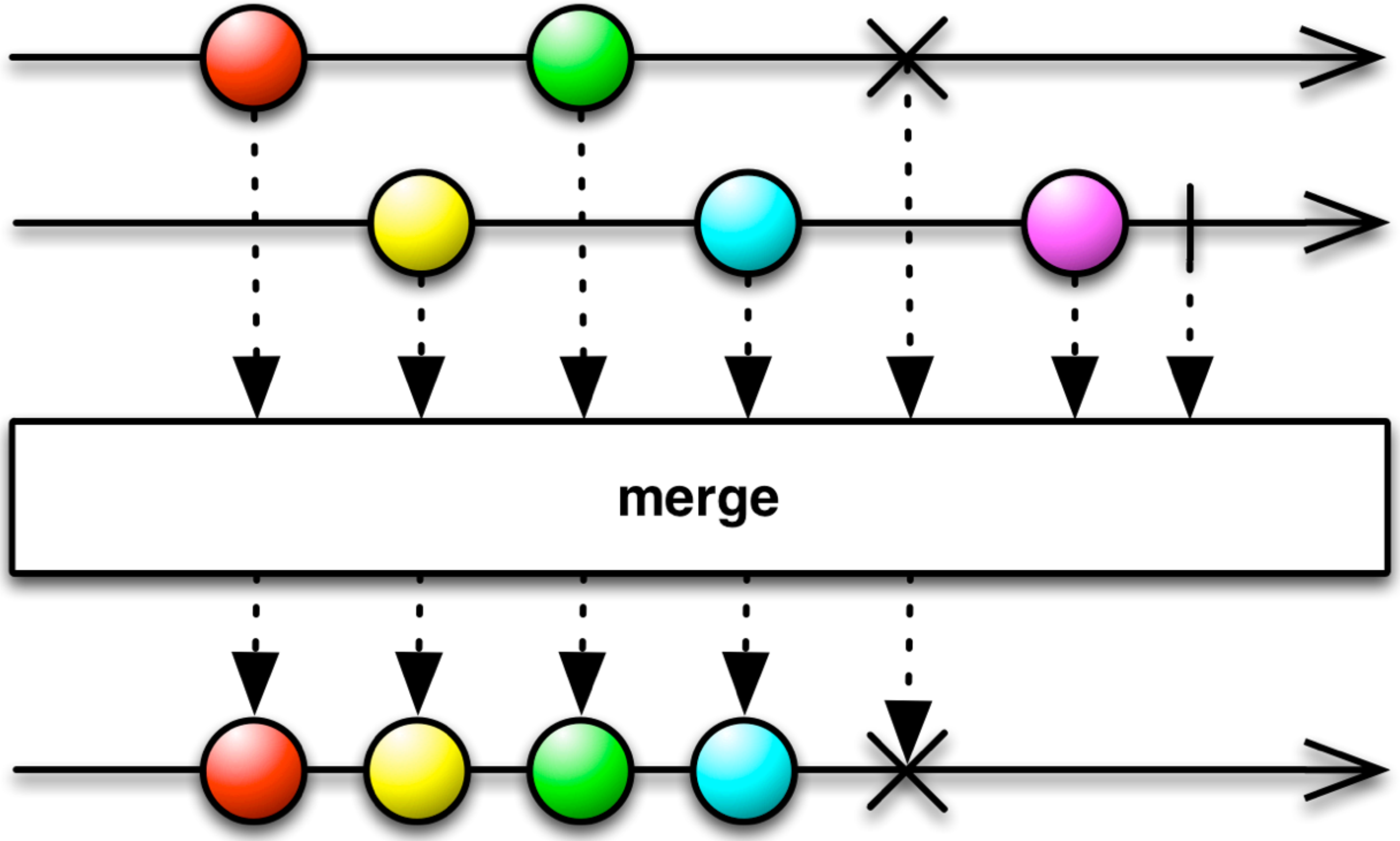


The ‘merge’ operator is used to combine multiple Observable sequences of the same type into a single Observable sequence with all data.

The X represents an onError call that would terminate the sequence so once it occurs the merged Observable also ends. The ‘mergeDelayError’ operator allows delaying the error until after all other values are successfully merged.



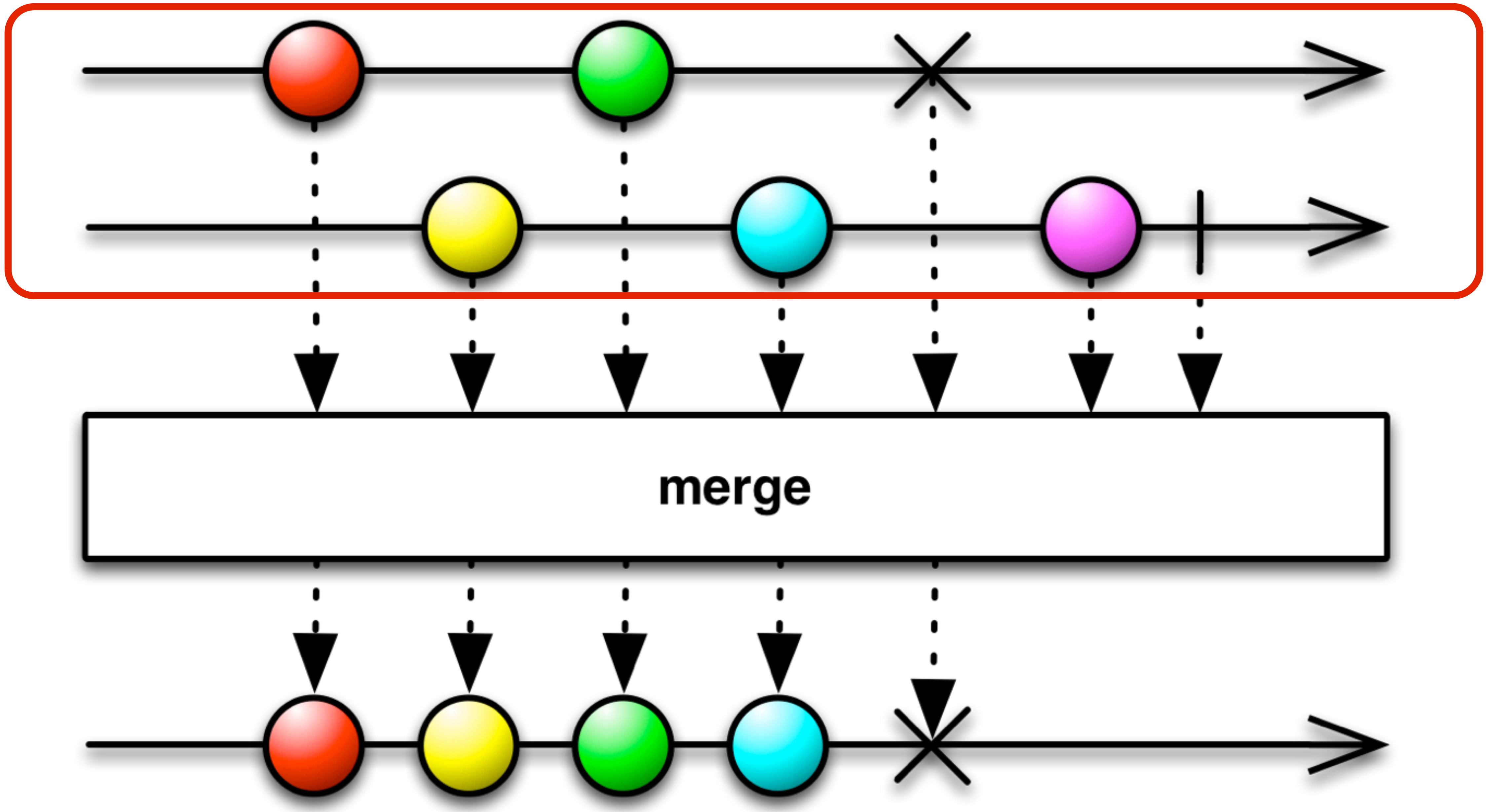

```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();

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




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

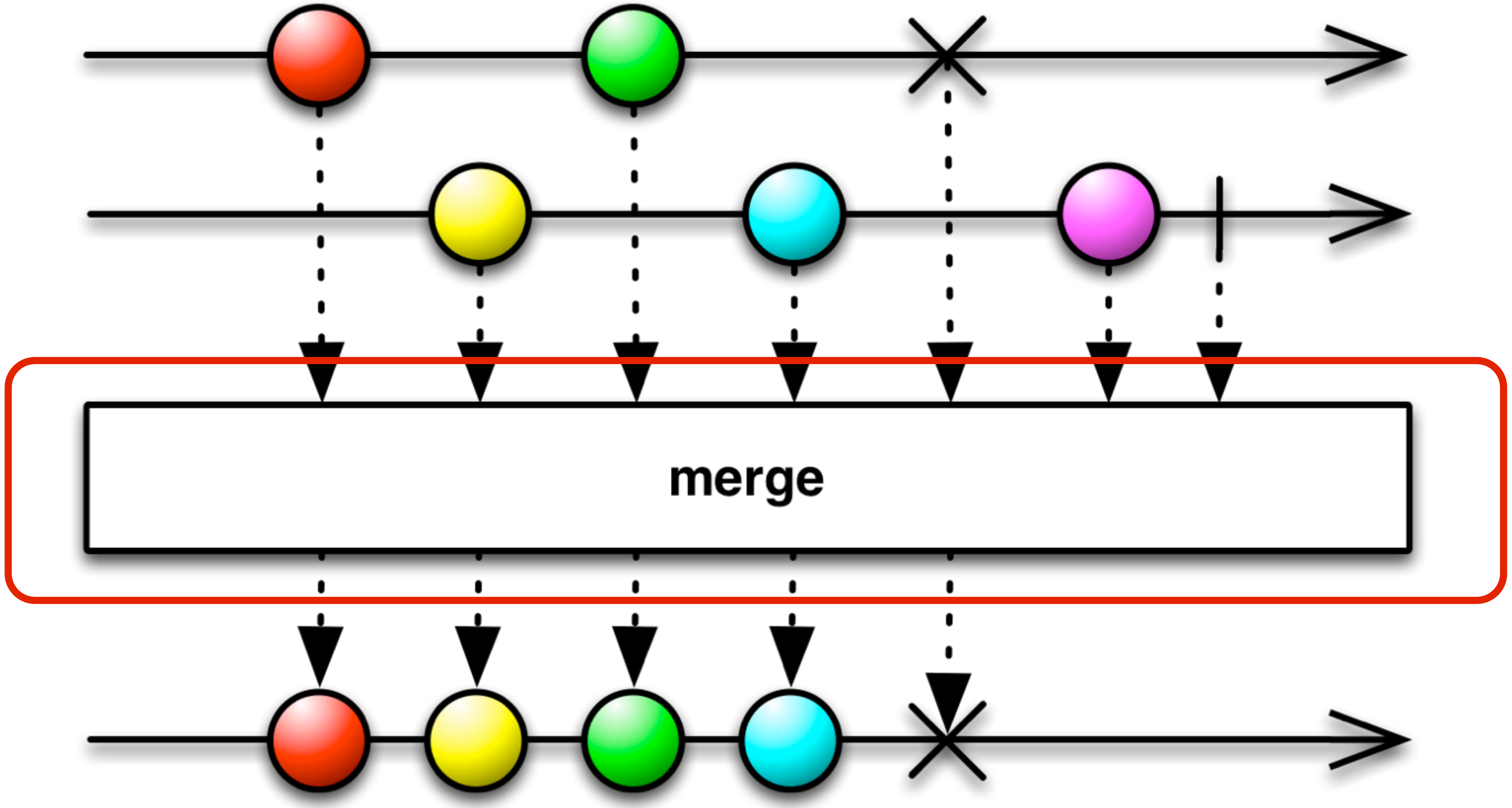
```
Observable.merge(a, b)
    .subscribe(
        { element -> println("data: " + element)})
```

Each of these Observables are of the same type...



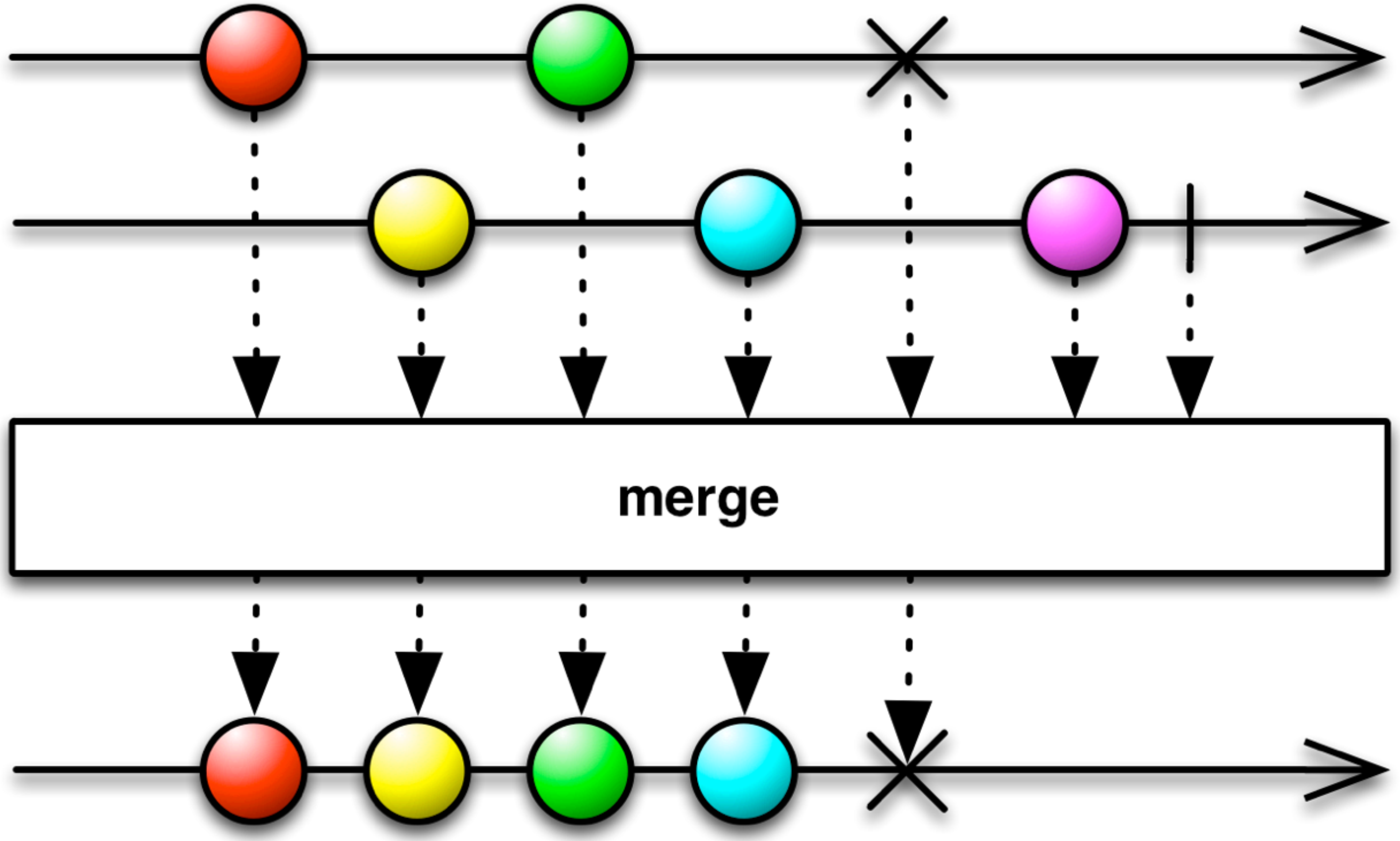

```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();

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

... and can be represented by these timelines ...



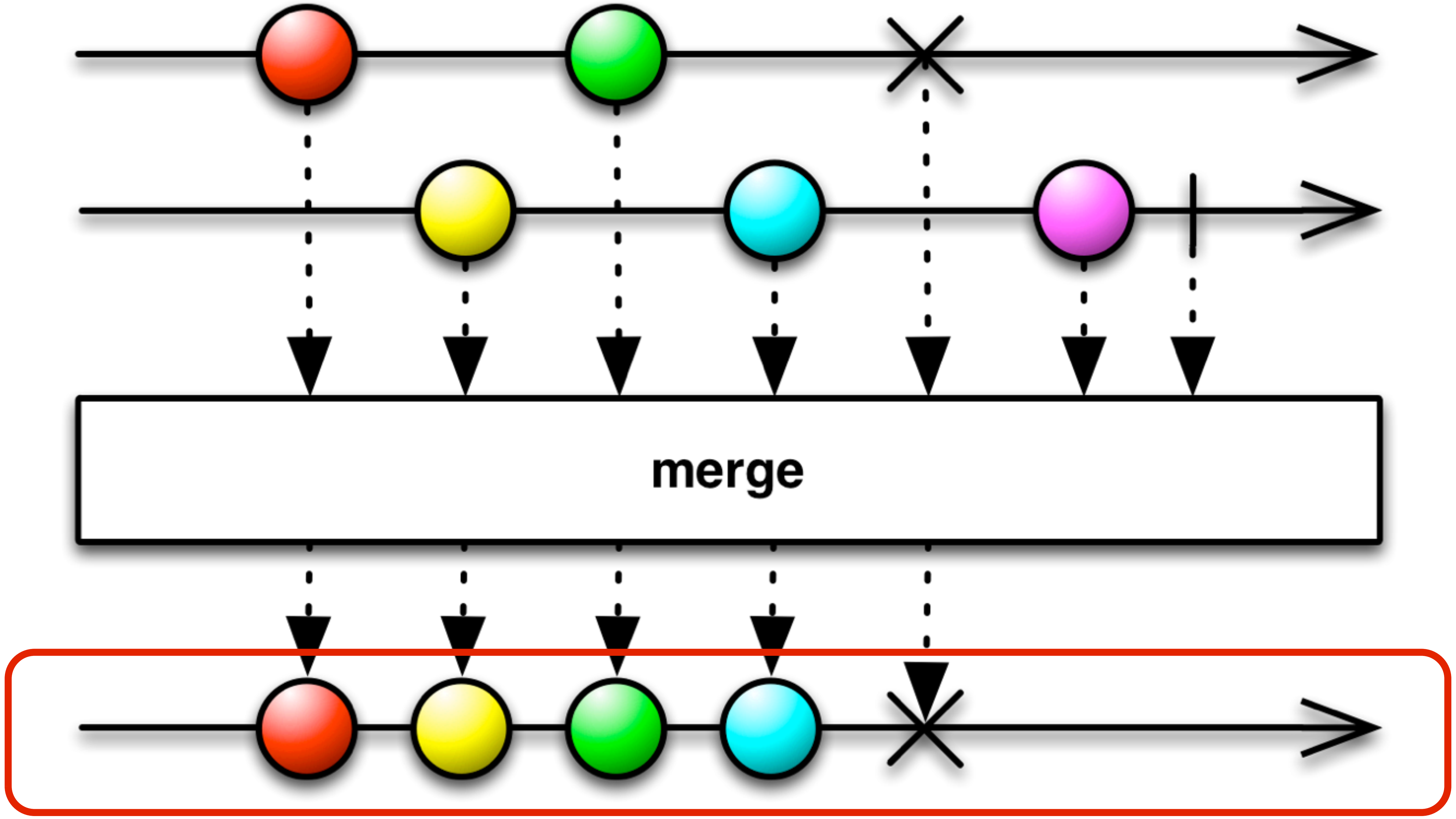

```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();

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

... that we pass through the ‘merge’ operator ...



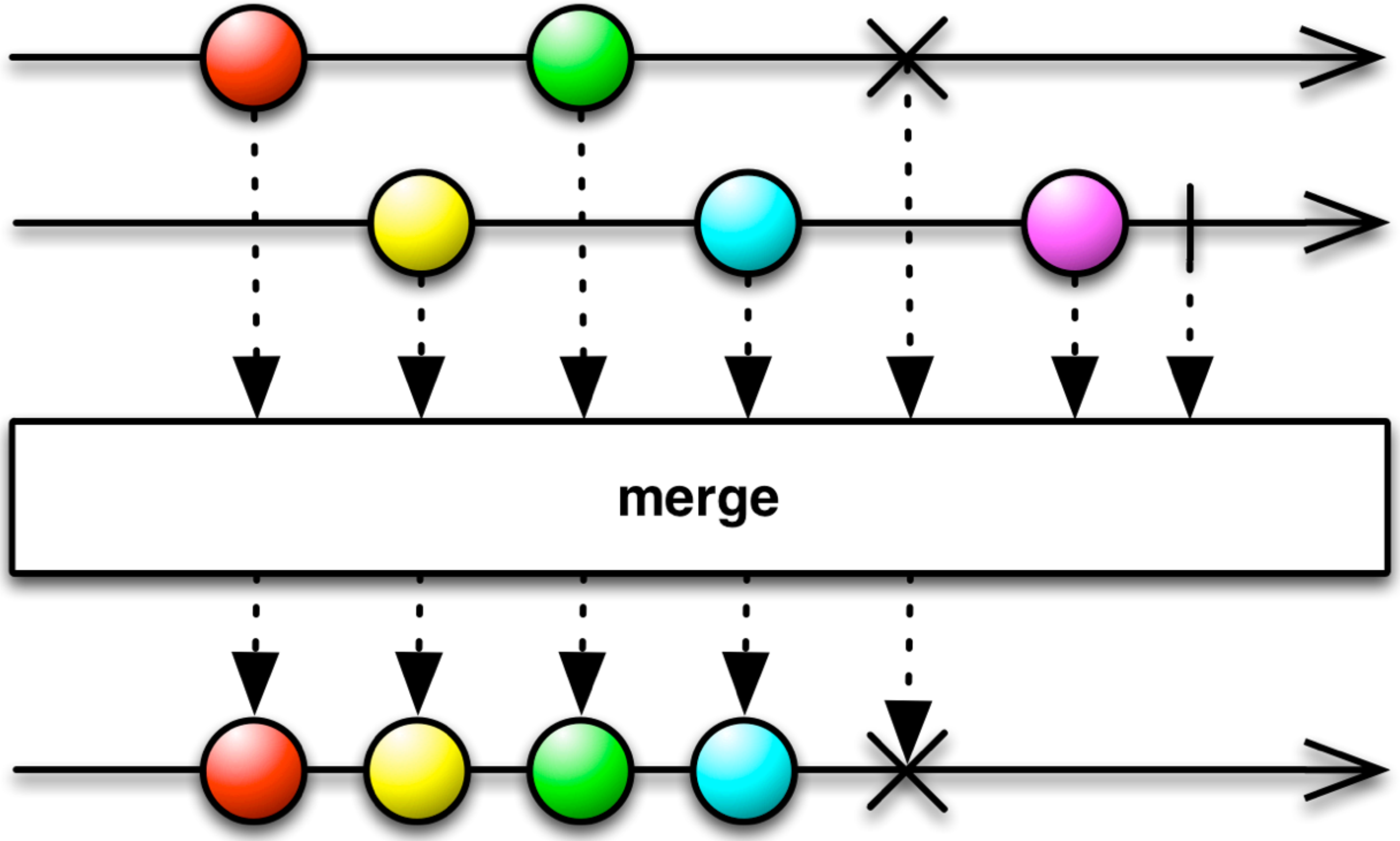

```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();

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

... which looks like this in code ...




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

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

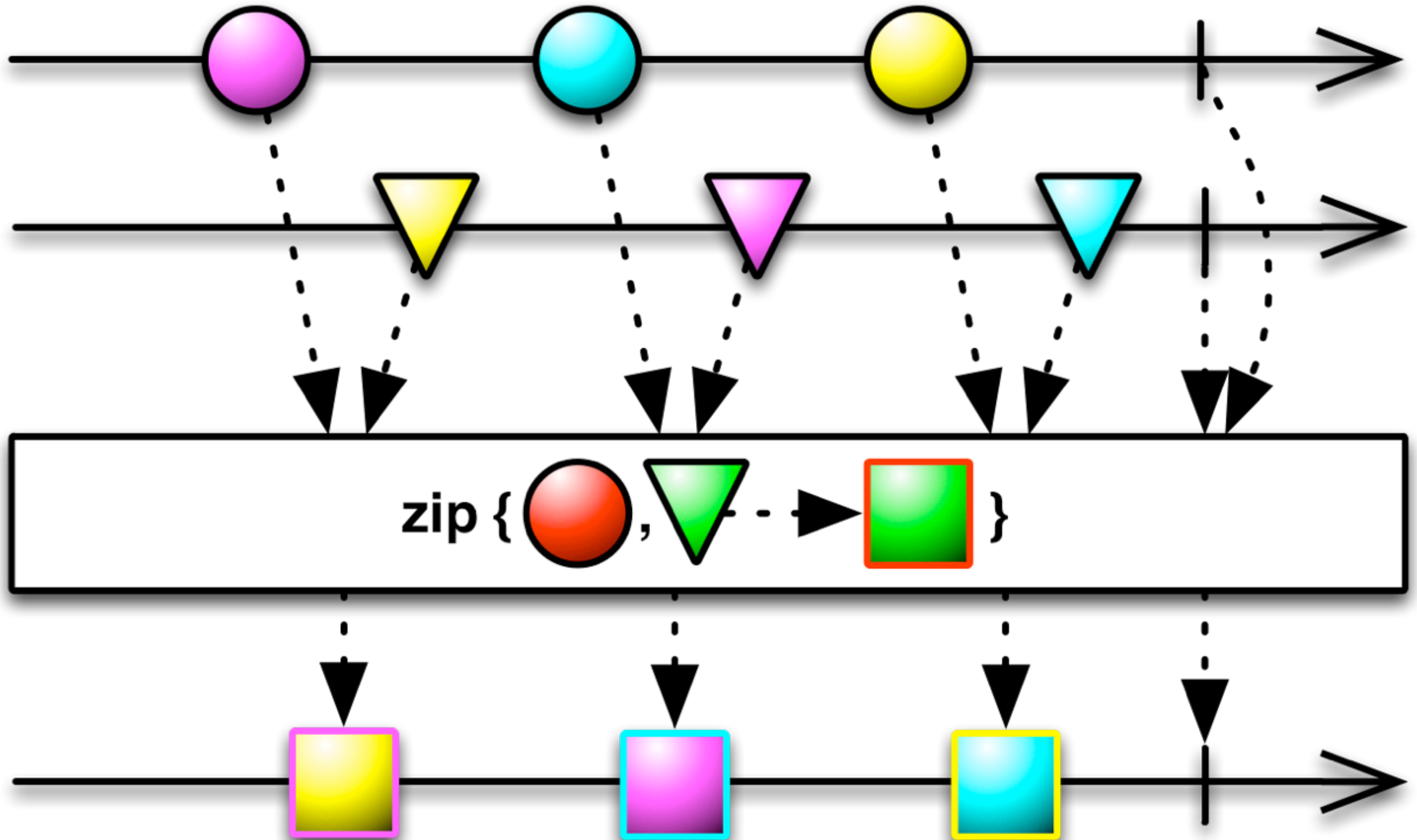
... and emits a single Observable containing all of the onNext events plus the first terminal event (onError/onCompleted) from the source Observables ...



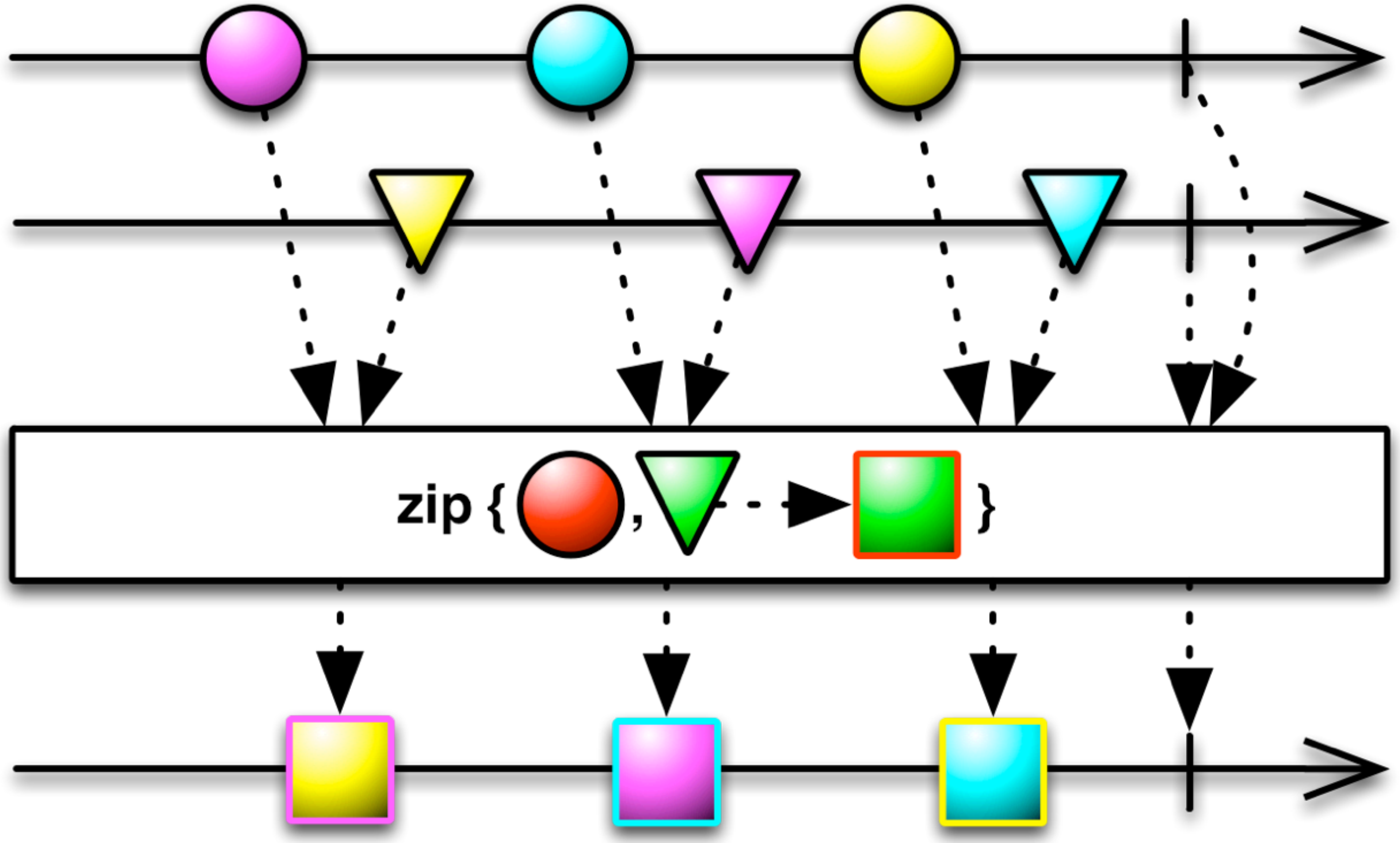

```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();

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

... and these are then subscribed to as a single Observable.

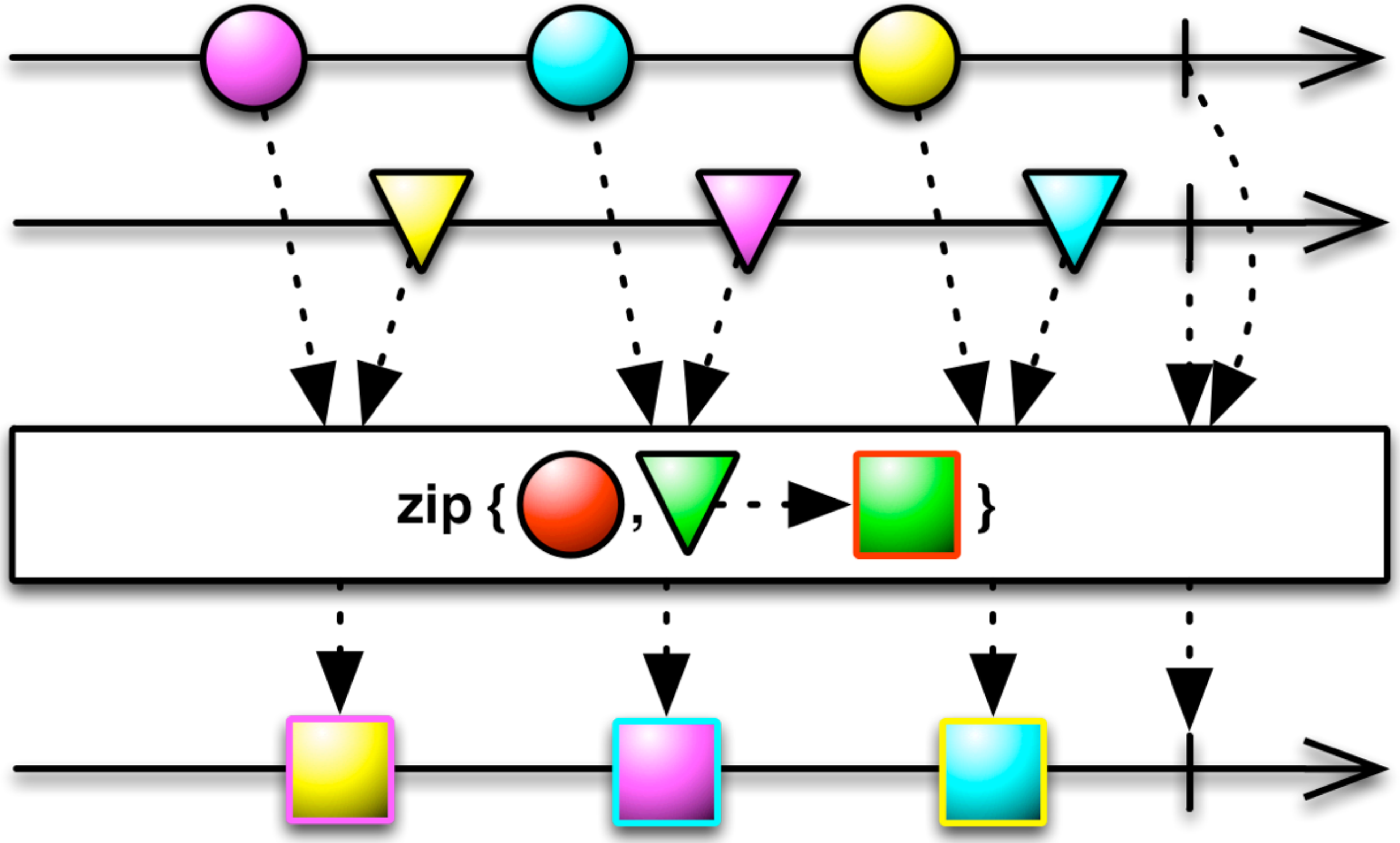
Combining via Zip



The ‘zip’ operator is used to combine Observable sequences of different types.



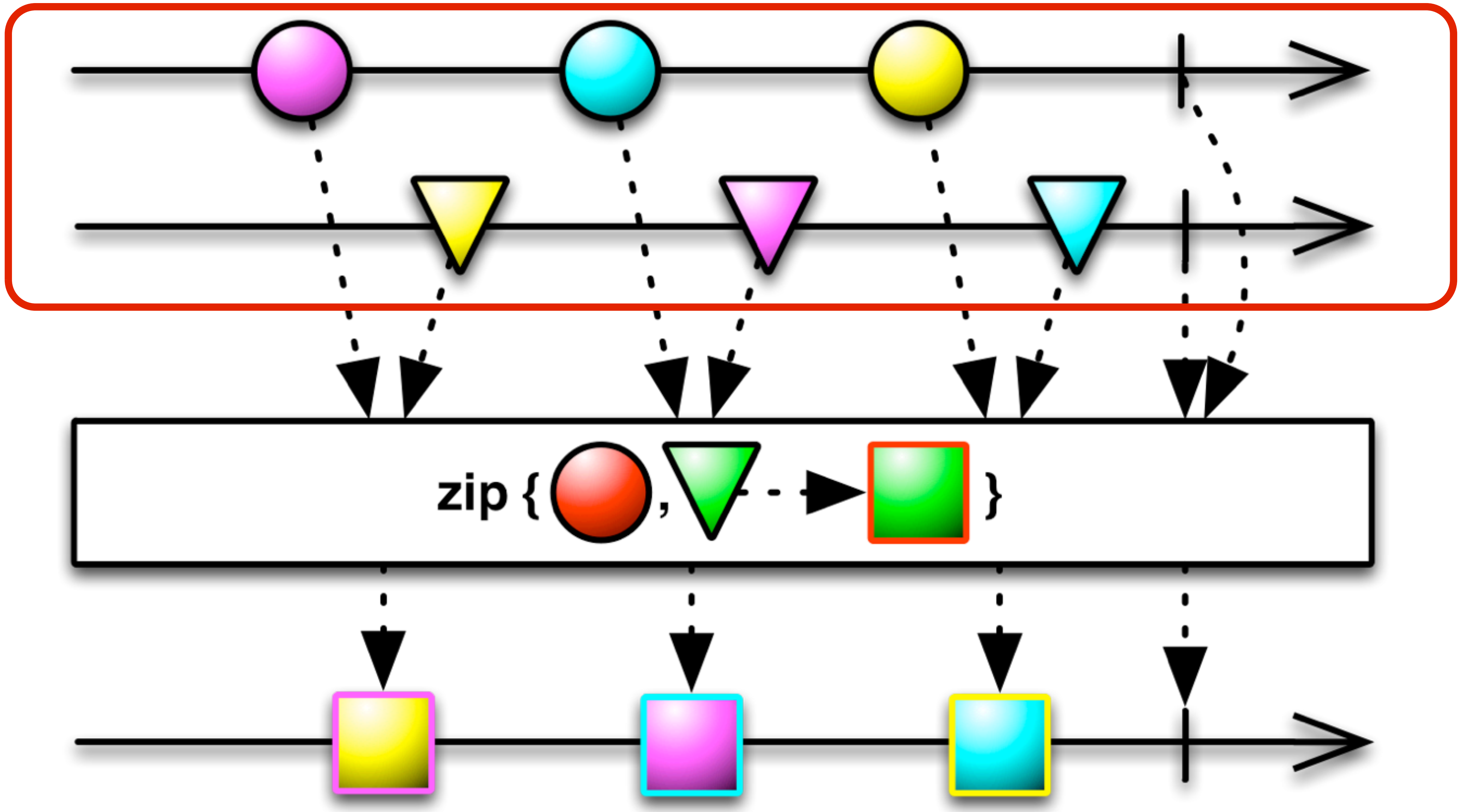

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

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();
```

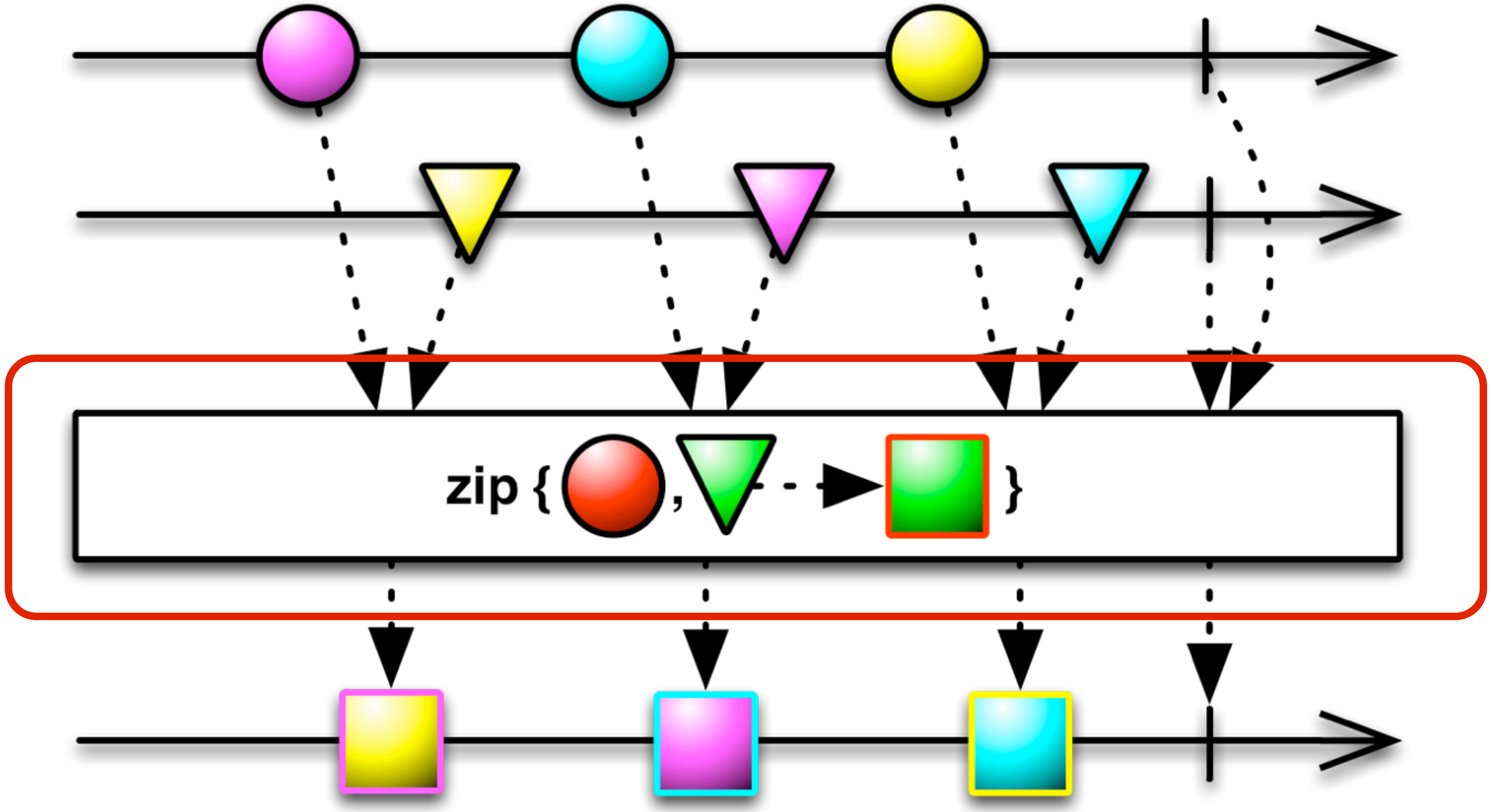
```
Observable.zip(a, b, {x, y -> [x, y]})
    .subscribe(
        { pair -> println("a: " + pair[0]
                        + " b: " + pair[1])})
```

Here are 2 Observable sequences with different types ...



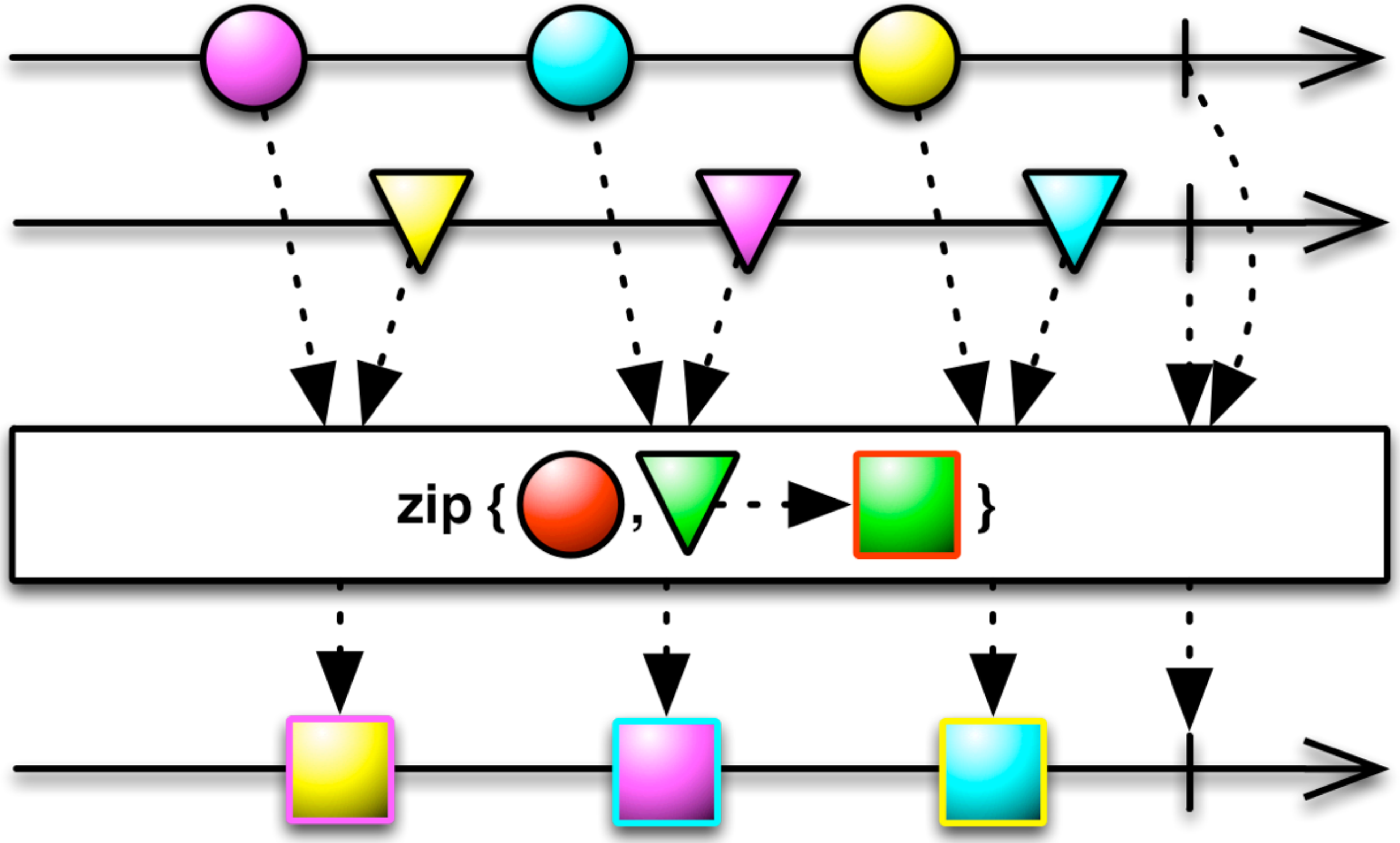

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

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

... represented by 2 timelines with different shapes ...



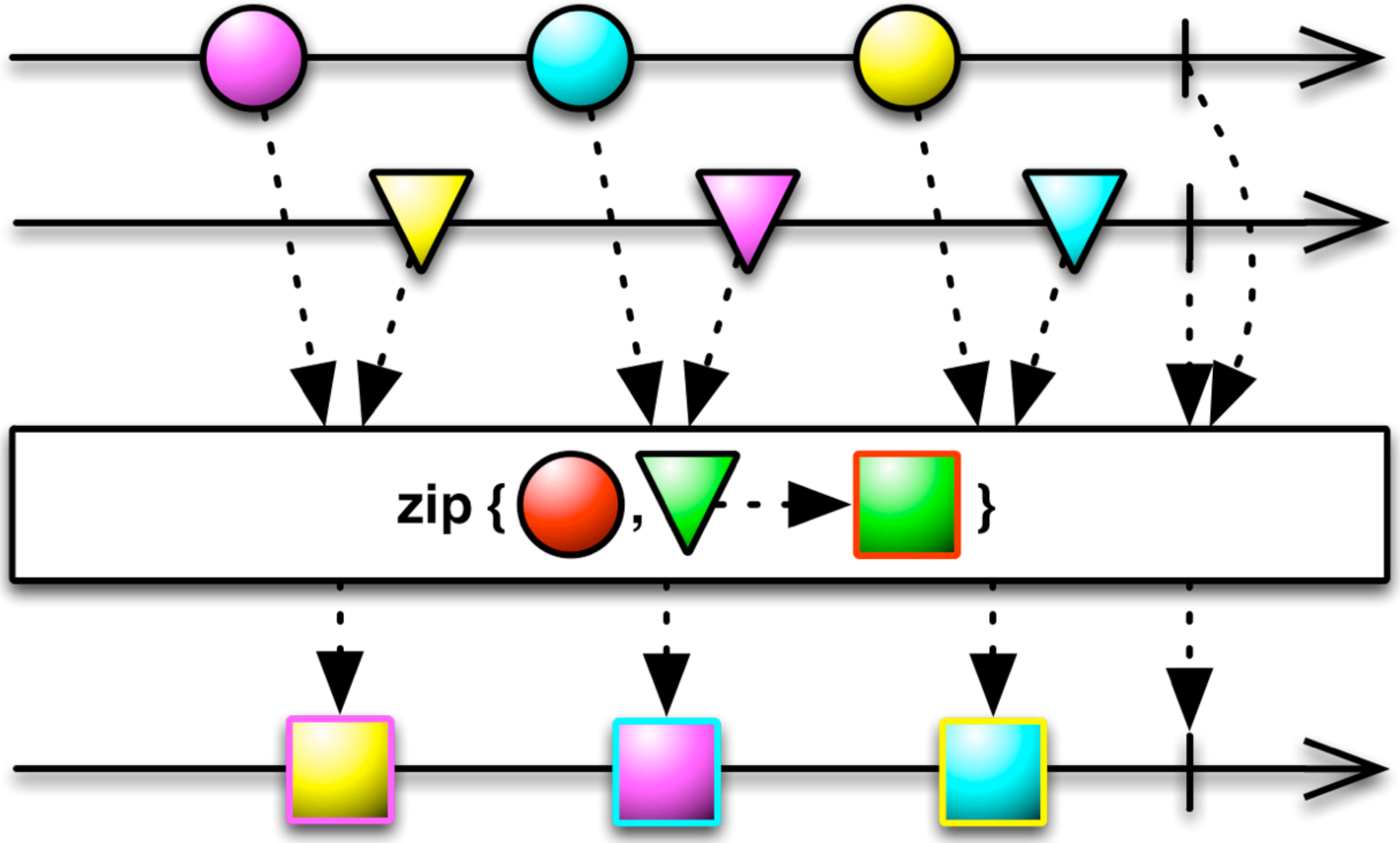

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

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

... that we pass through the zip operator that contains a provided function to apply to each set of values received.



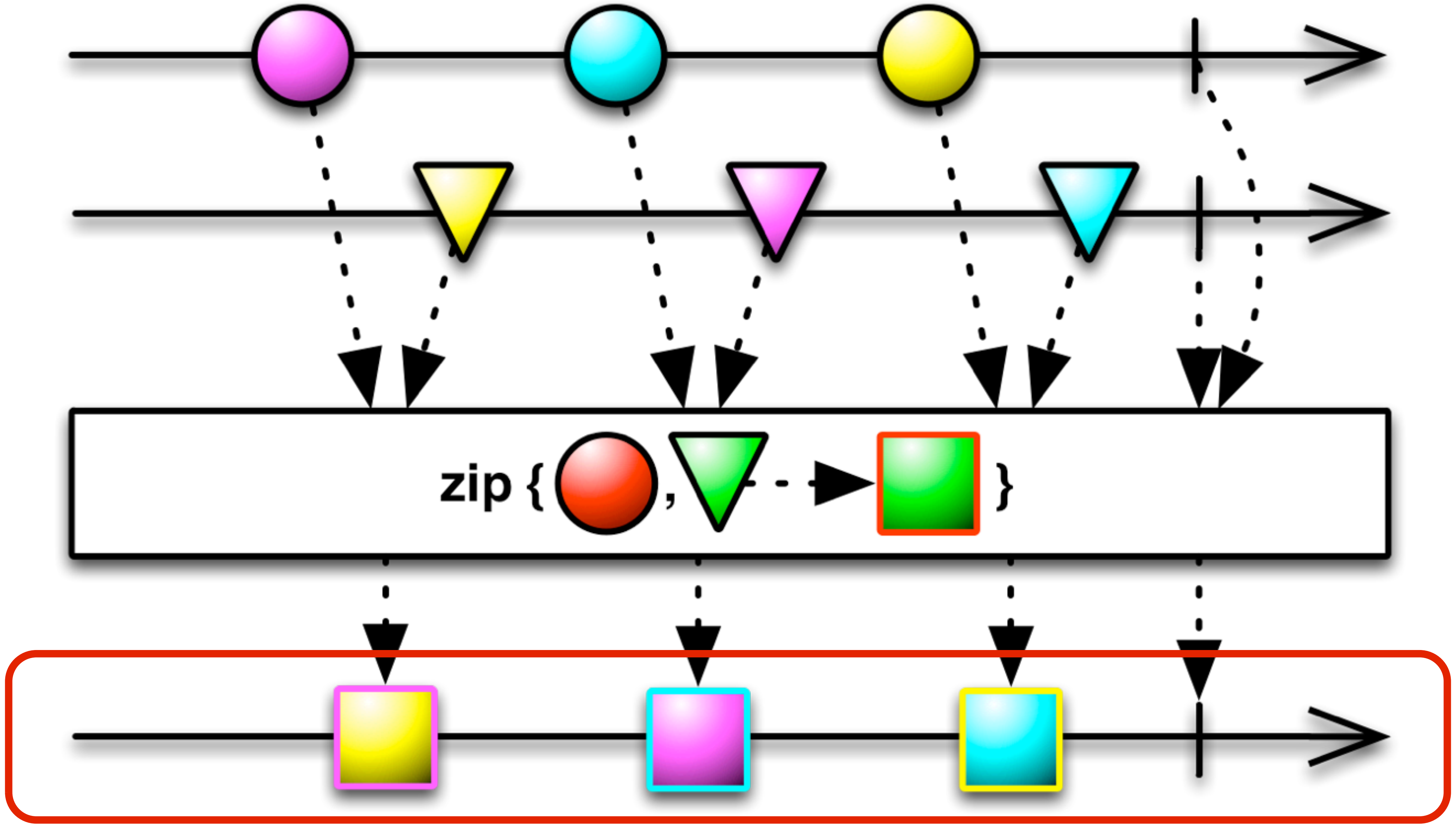

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

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

The transformation function is passed into the zip operator ...



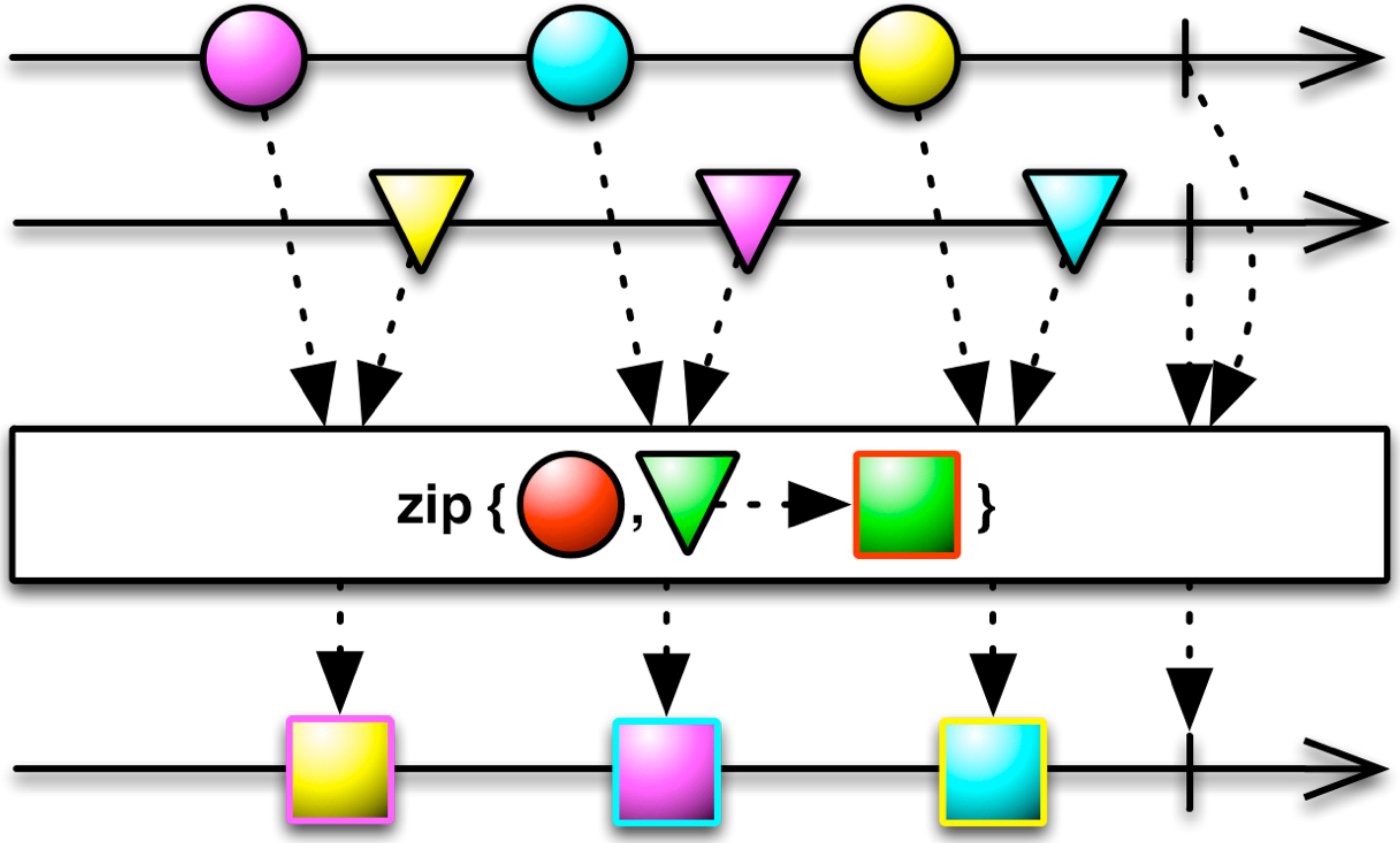

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();

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

... and in this case is simply taking x & y and combining them into a tuple or pair and then returning it.




```
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])})
```

The output of the transformation function given to the zip operator is emitted in a single Observable sequence ...




```
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])})
```

... that gives us our pairs when we subscribe to it.

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") })
```

If an error occurs then the ‘onError’ handler passed into the ‘subscribe’ will be invoked...

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") })
```

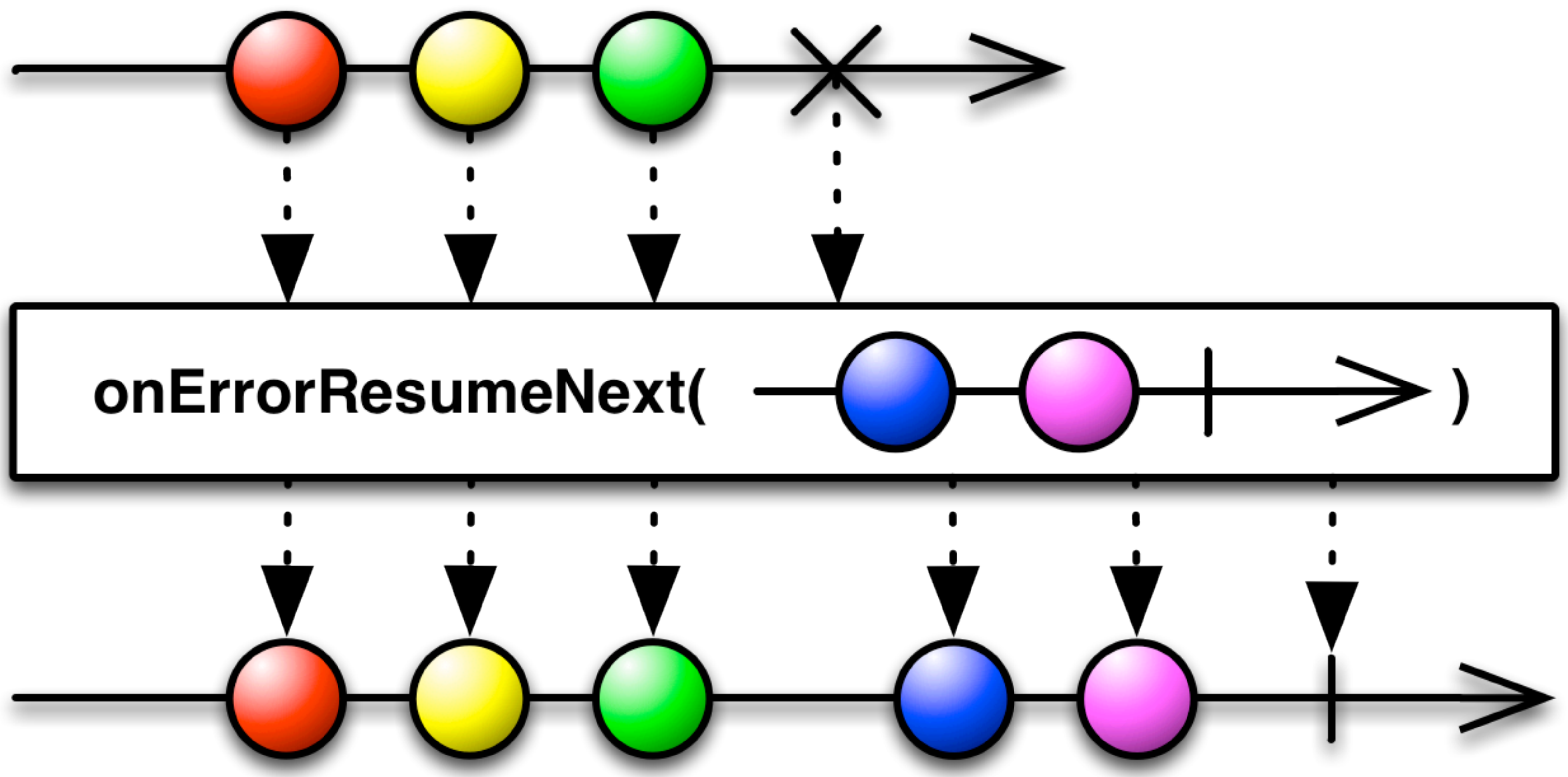
onNext(T)

onError(Exception)

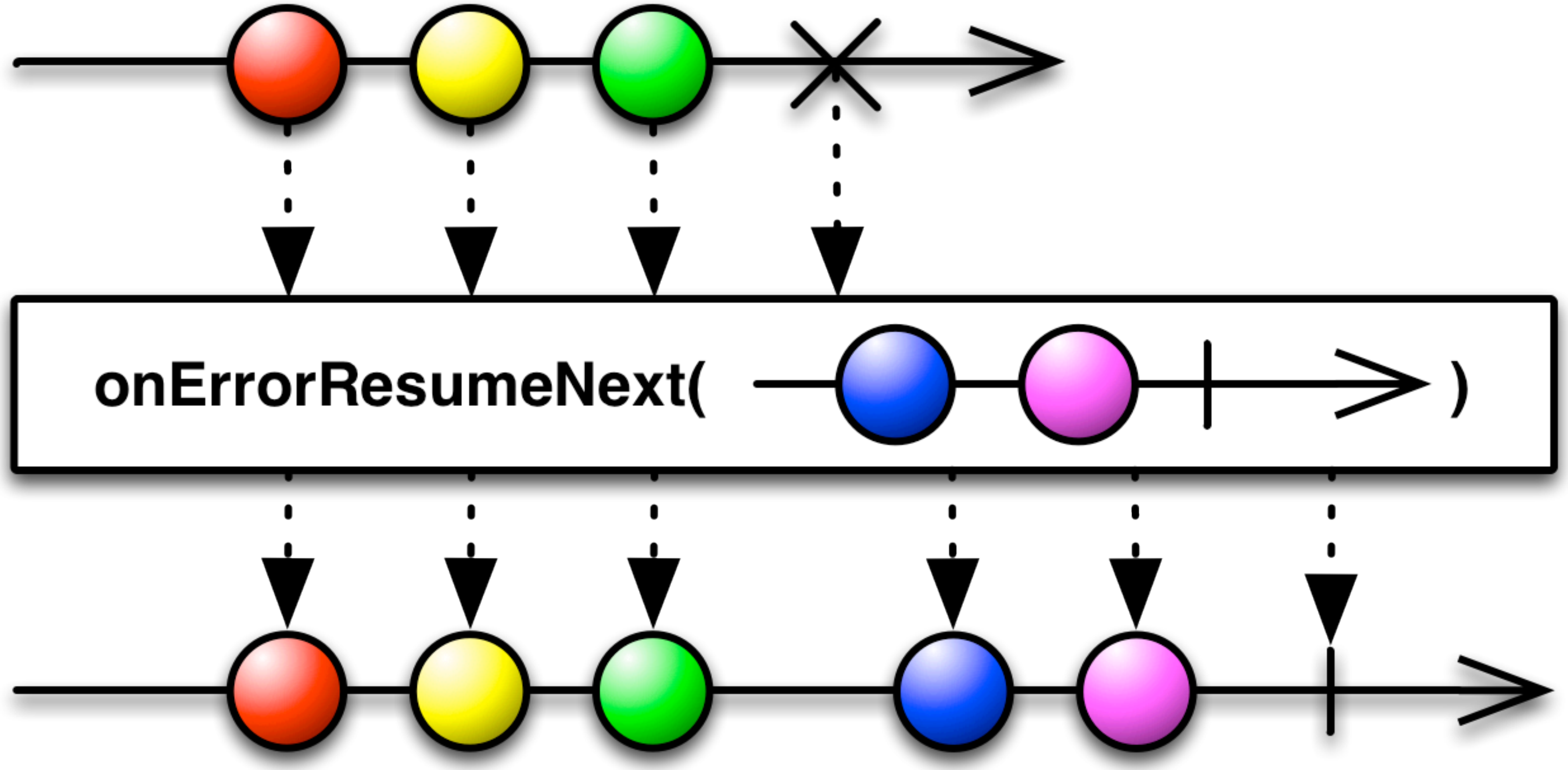
onCompleted()

... but this is the final terminal state of the entire composition so we often want to move our error handling to more specific places. There are operators for that ...

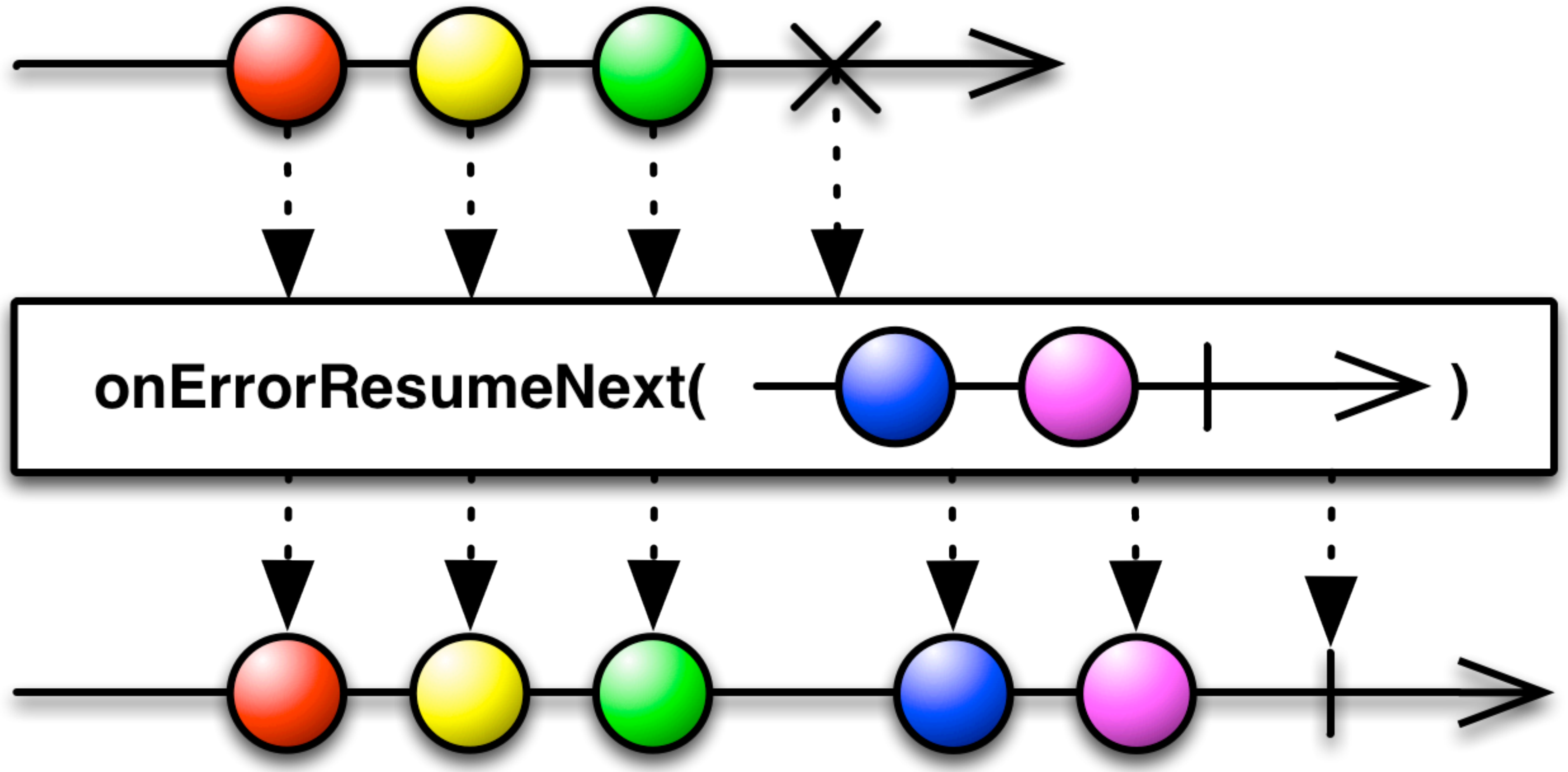
Error Handling



The ‘onErrorResumeNext’ operator allows intercepting an ‘onError’ and providing a new Observable to continue with.

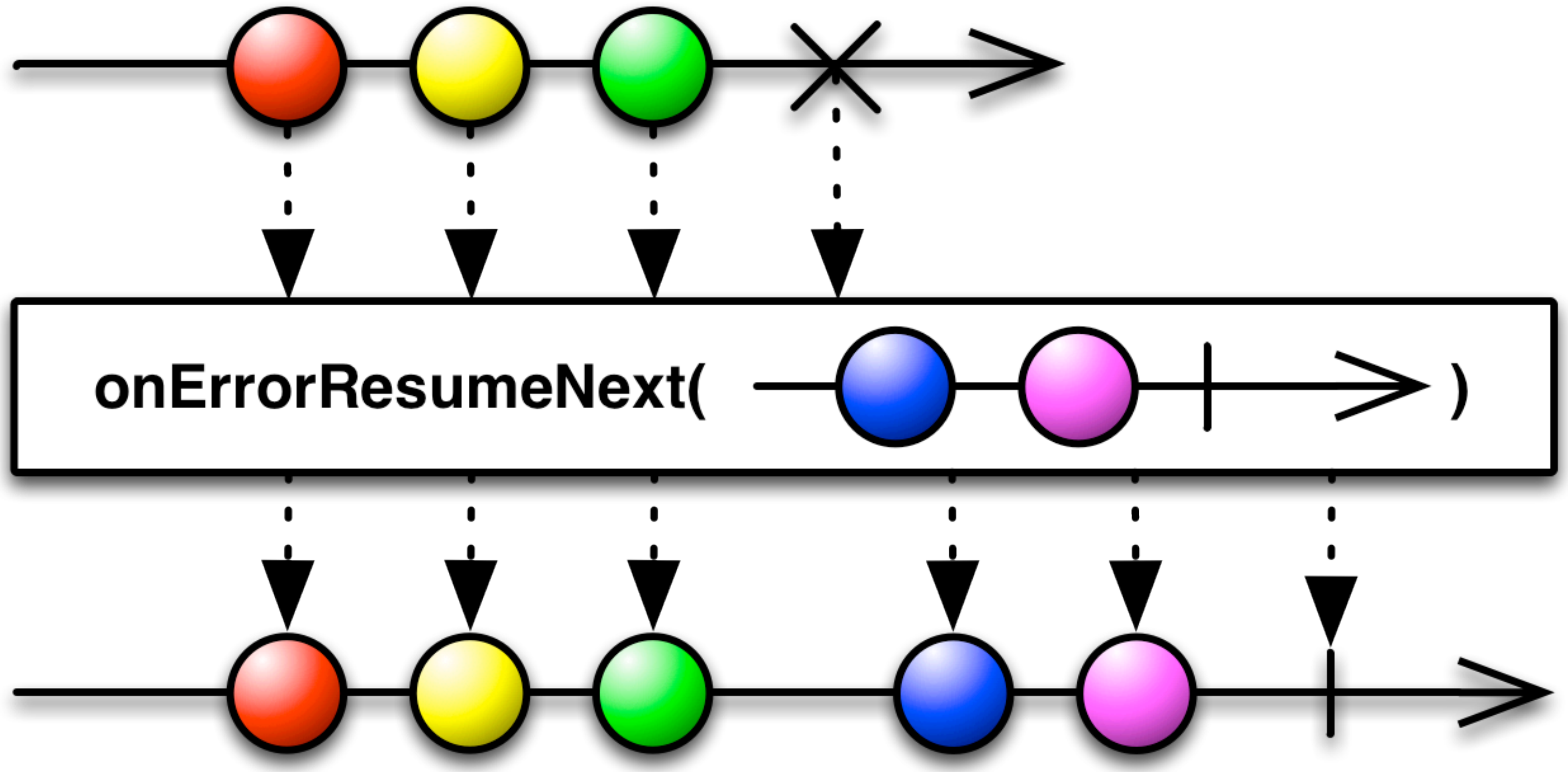



```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()


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



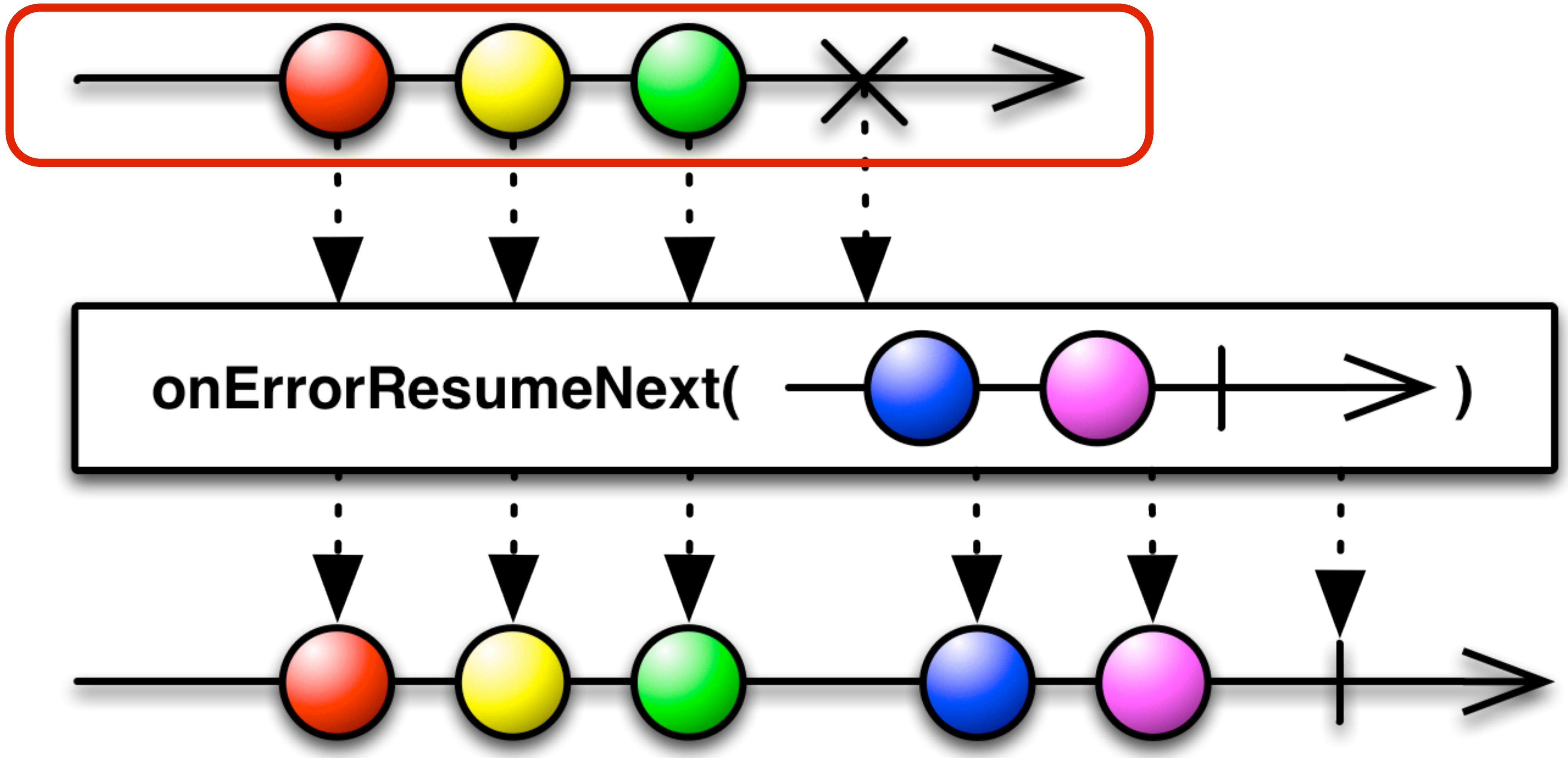

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                          .onErrorResumeNext(getFallbackForB());

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

If we want to handle errors on Observable ‘b’ we can compose it with ‘onErrorResumeNext’ and pass in a function that when invoked returns another Observable that we will resume with if onError is called.



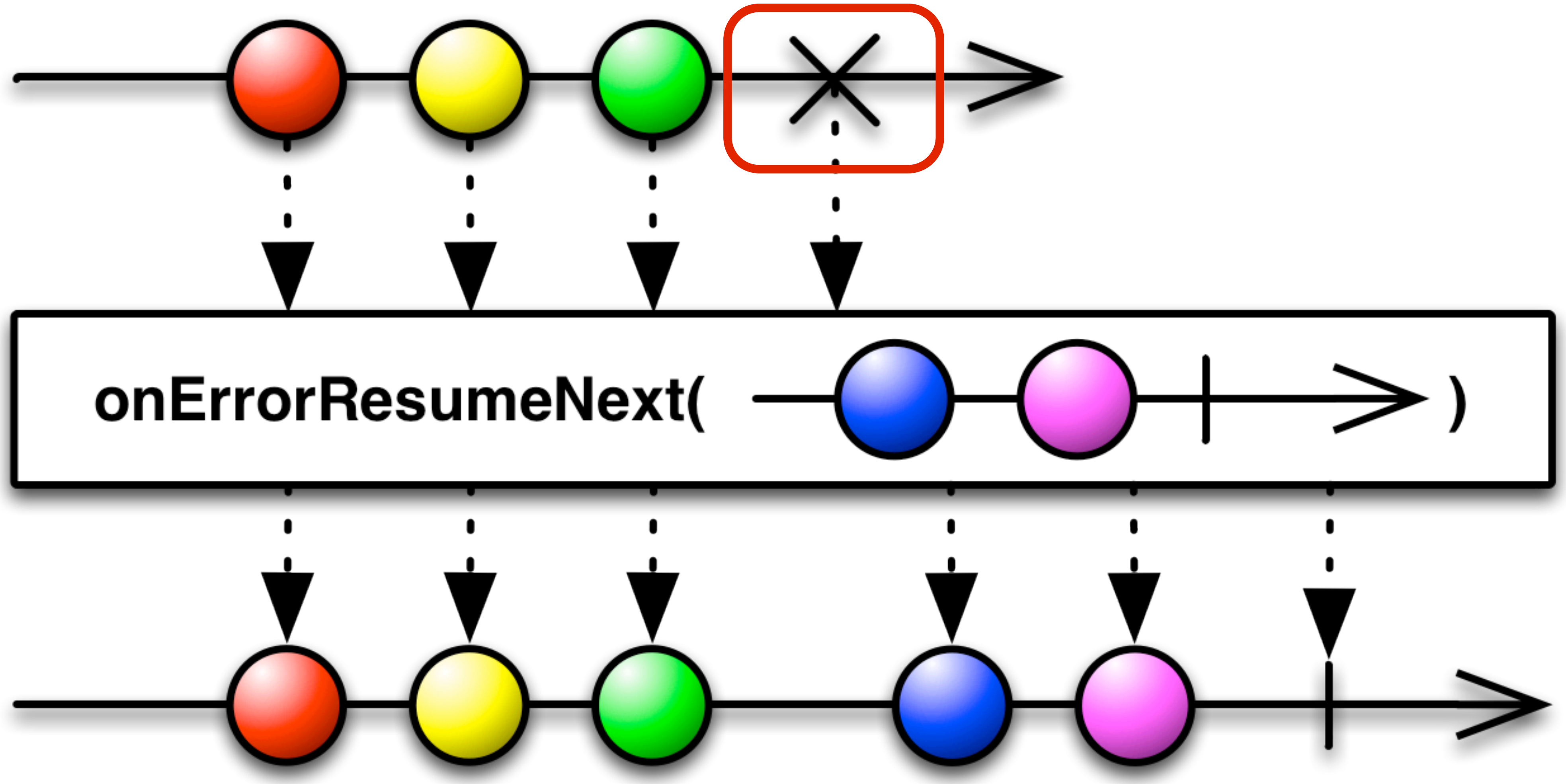

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                            .onErrorResumeNext(getFallbackForB());

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

So ‘b’ represents an Observable sequence ...



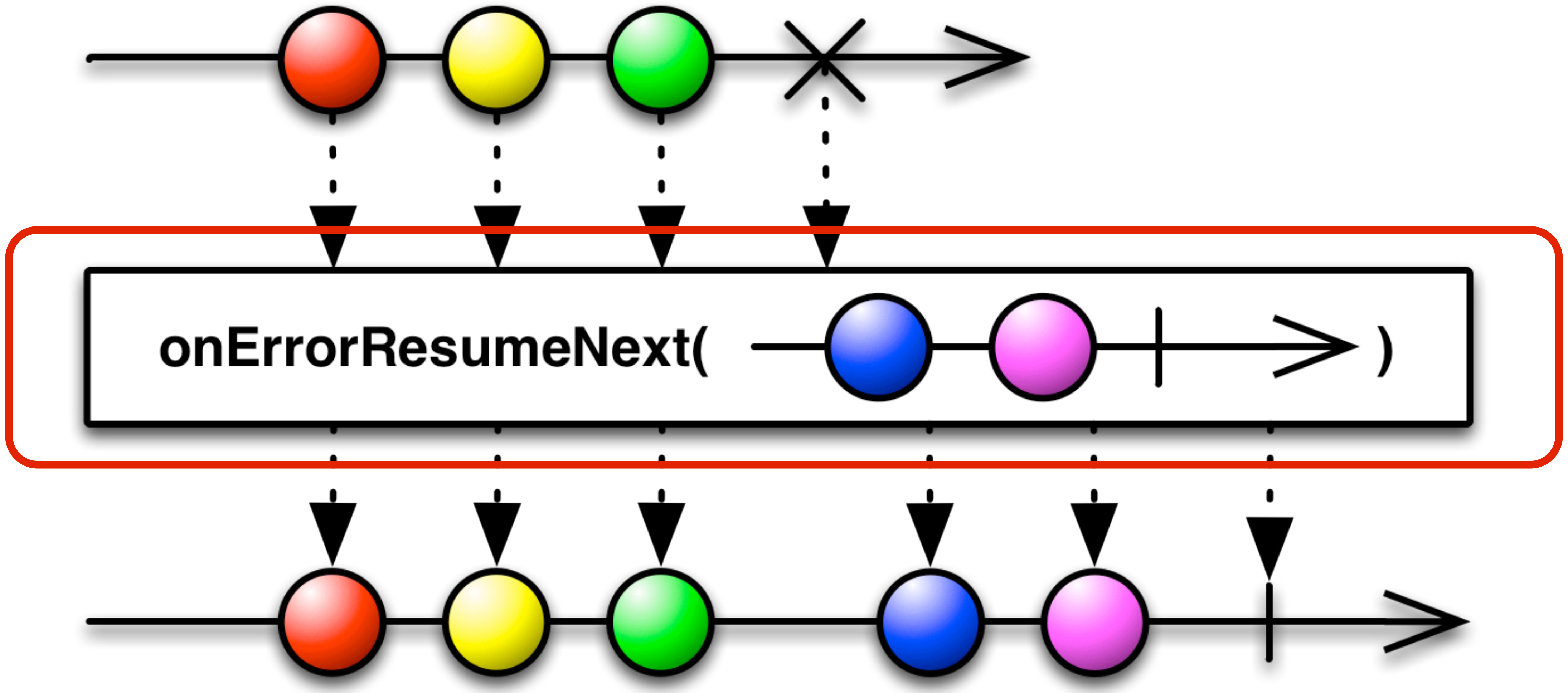

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                          .onErrorResumeNext(getFallbackForB());

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

... that emits 3 values ...



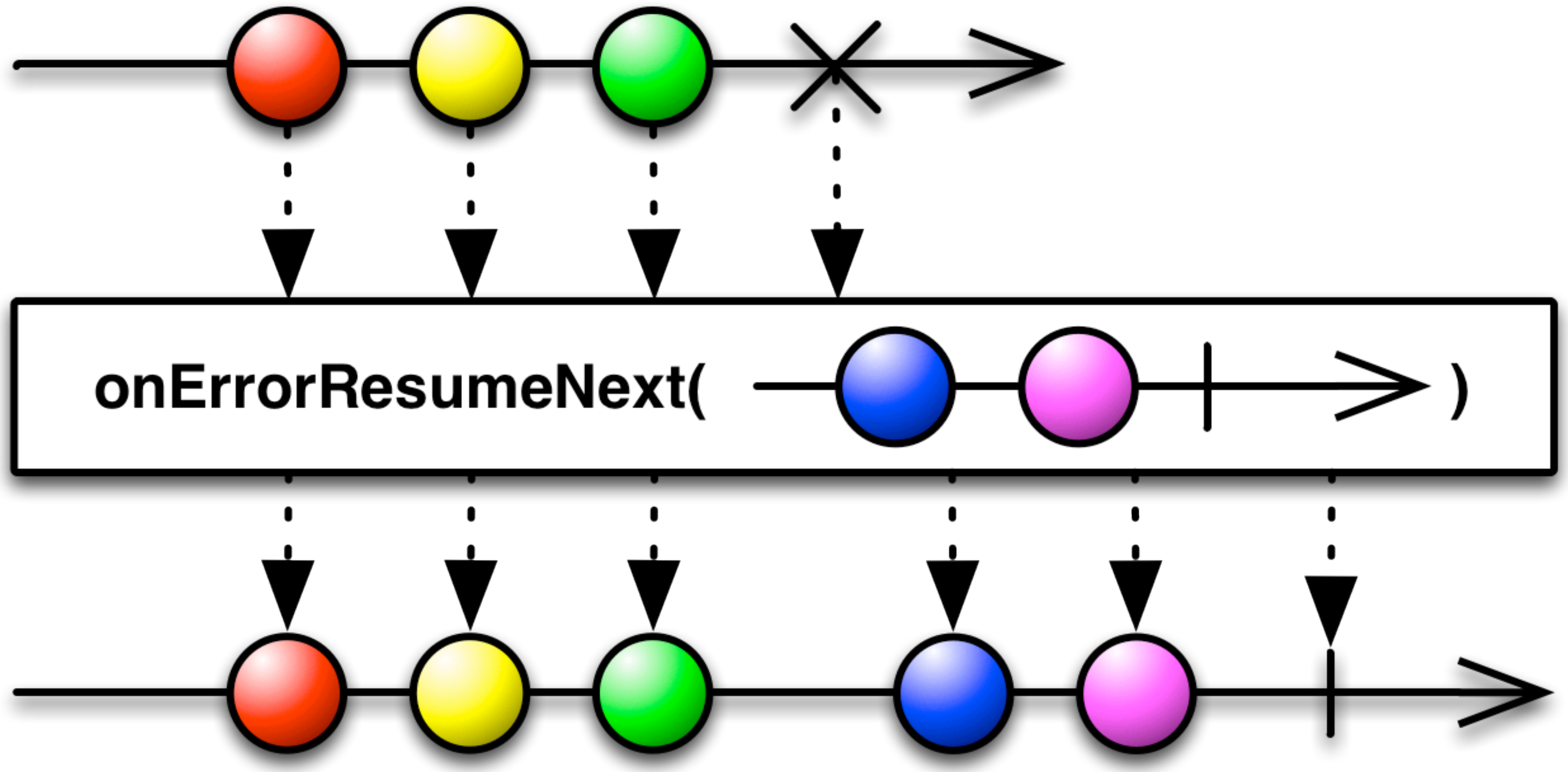

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                          .onErrorResumeNext(getFallbackForB());

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

... and then fails and calls onError ...



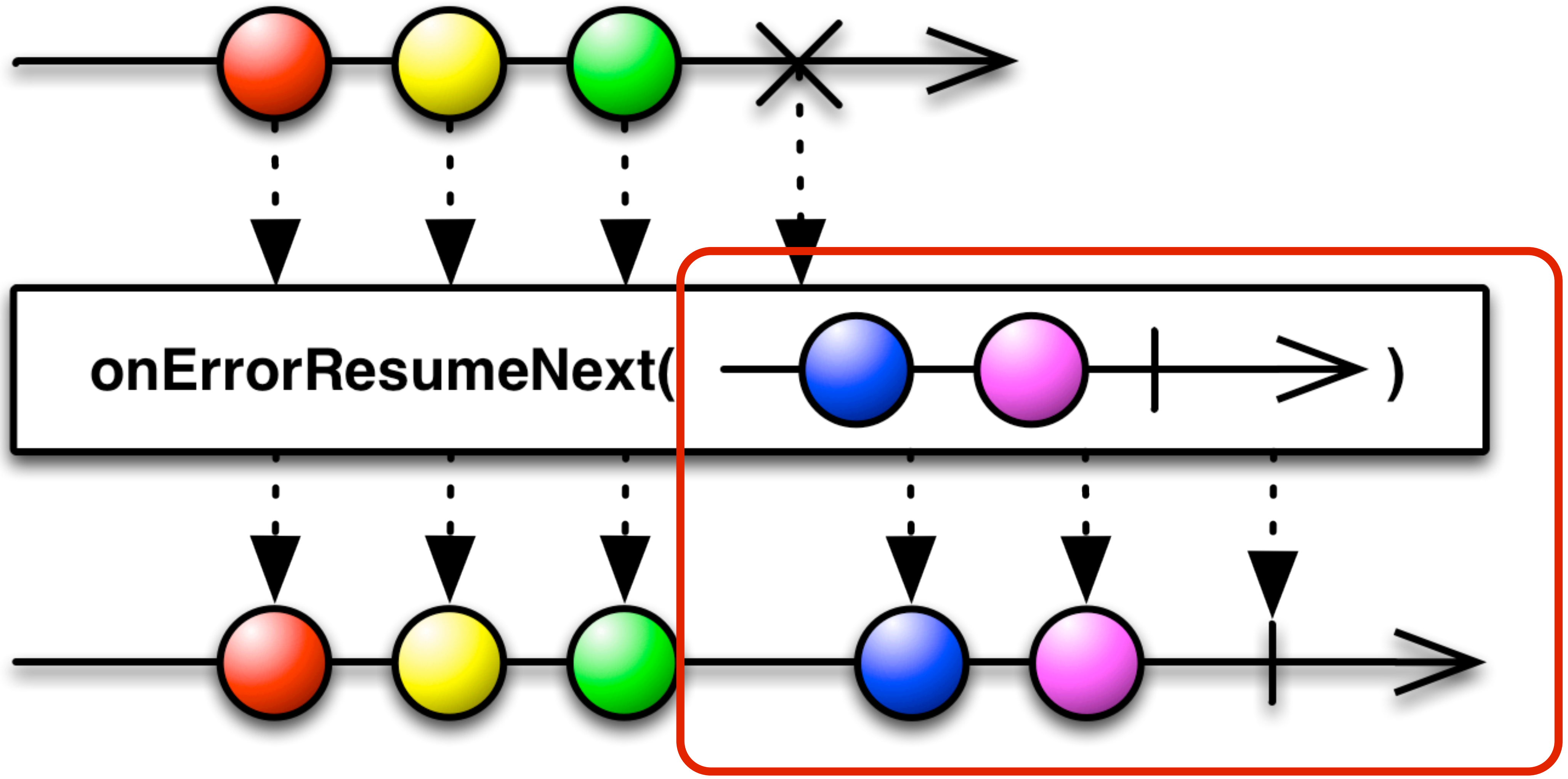

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                          .onErrorResumeNext(getFallbackForB());

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

... which being routed through ‘onErrorResumeNext’ ...



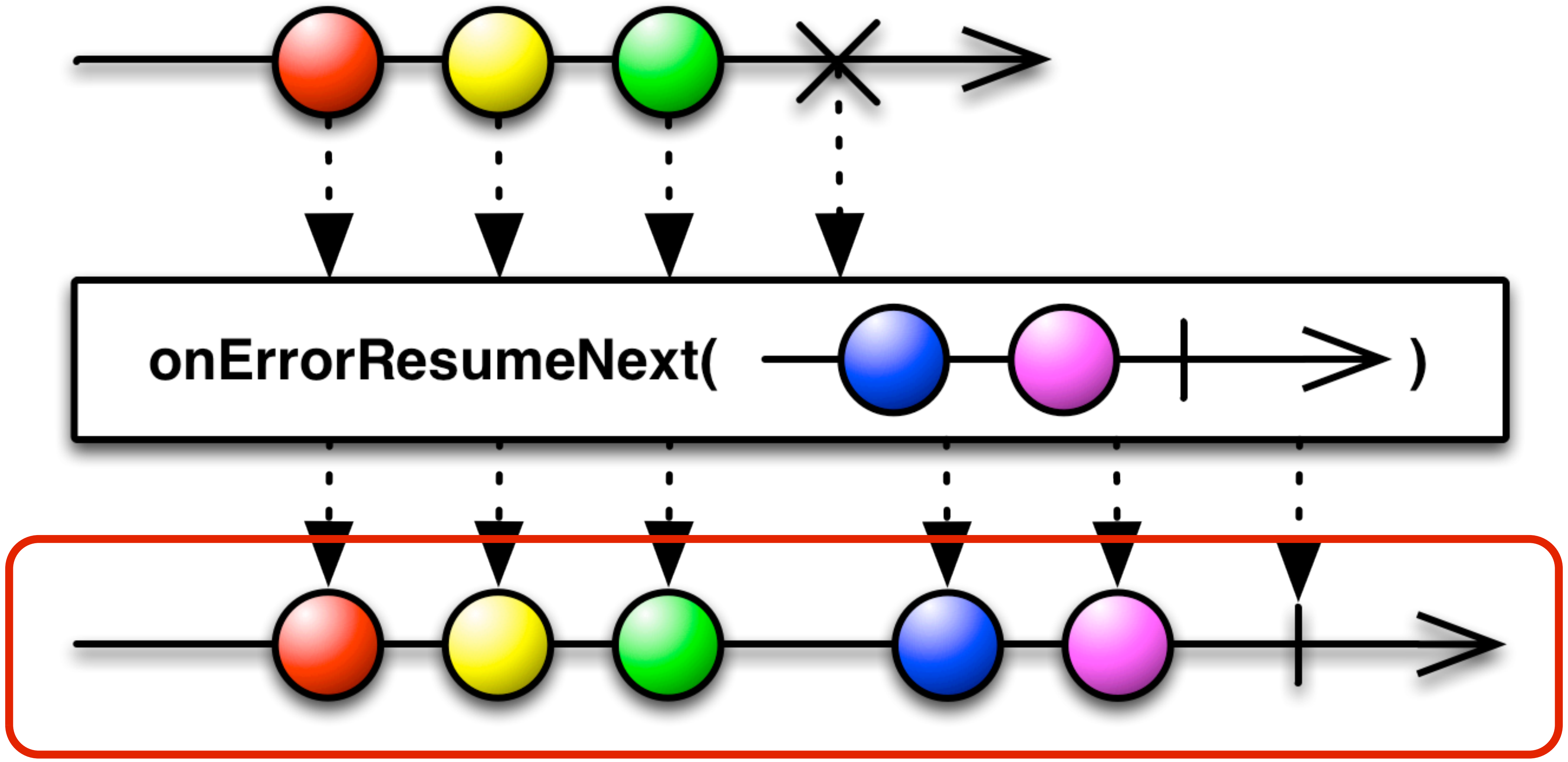

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                        .onErrorResumeNext(getFallbackForB());

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

... triggers the invocation of ‘getFallbackForB()’ ...



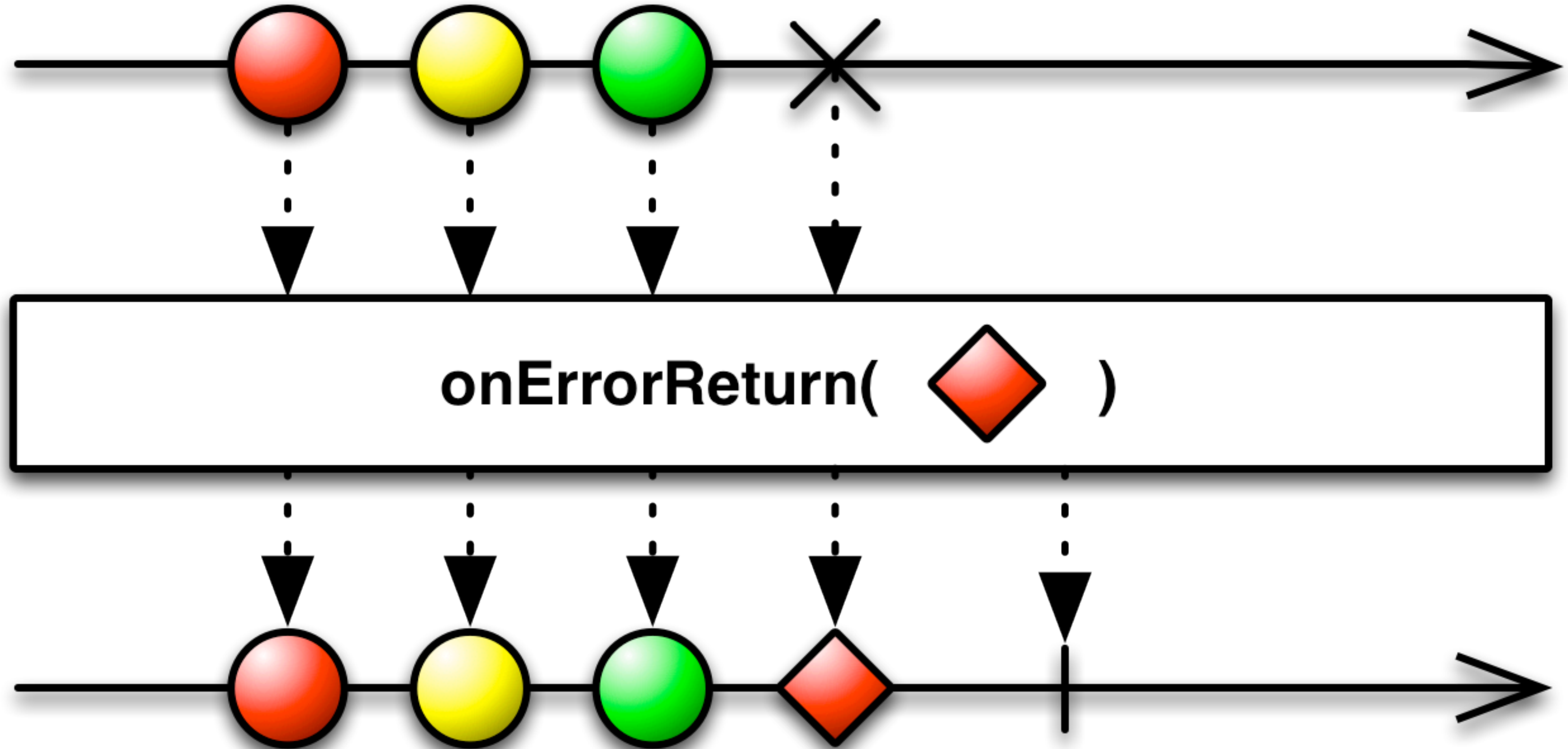

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                            .onErrorResumeNext(getFallbackForB());

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

... which provides a new Observable that is subscribed to in place of the original Observable ‘b’ ...

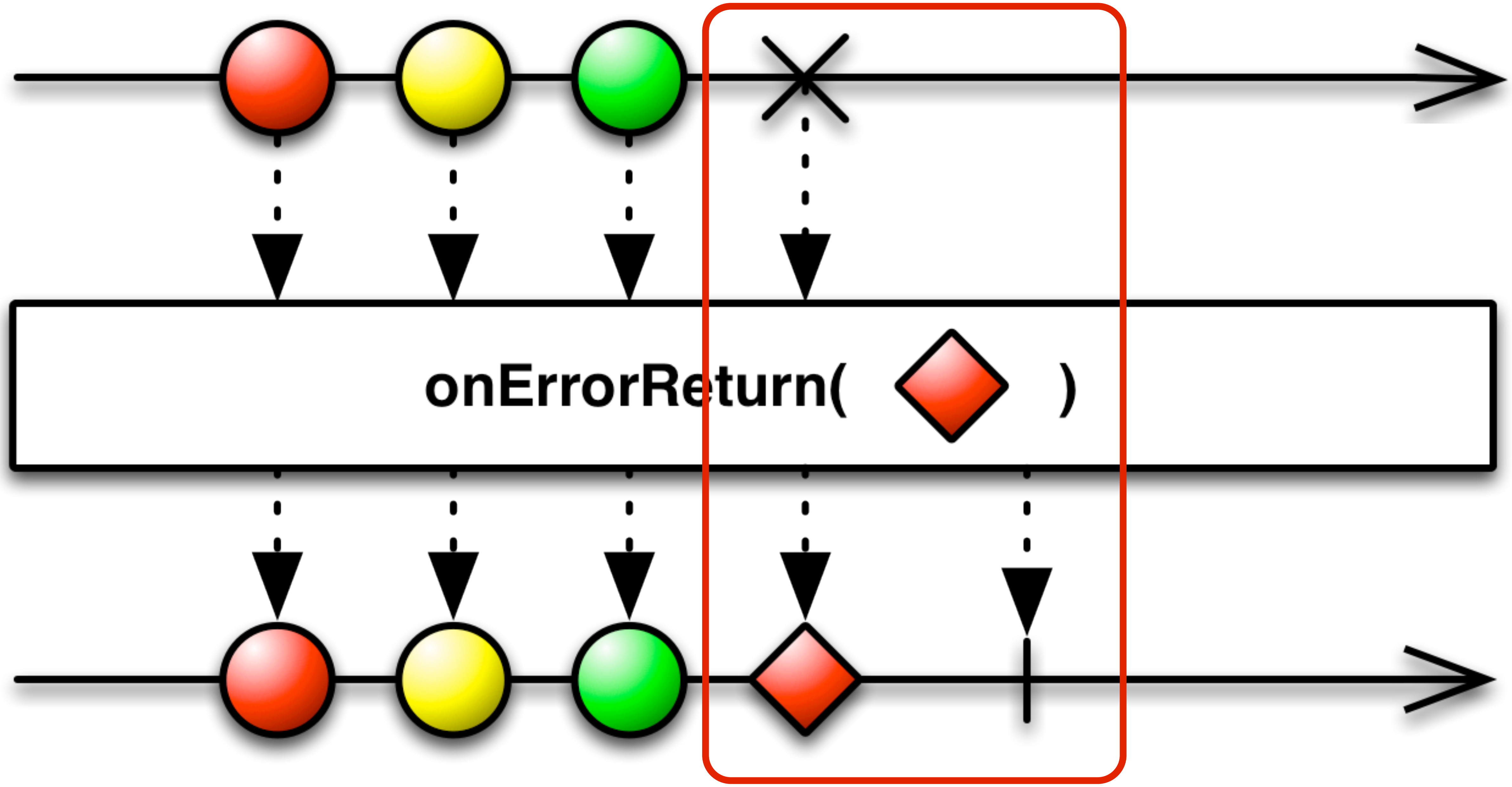



```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB()
                          .onErrorResumeNext(getFallbackForB());

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

... so the returned Observable emits a single sequence of 5 onNext calls and a successful onCompleted without an onError.



The ‘onErrorReturn’ operator is similar ...



... except that it returns a specific value instead of an Observable.

Various ‘onError*’ operators can be found in the Javadoc: http://netflix.github.com/RxJava/javadoc/rx/Observable.html

HTTP Request Use Case

```
ObservableHttp.createRequest(
   HttpAsyncMethods.createGet("http://www.wikipedia.com"), client)
```

HTTP requests will be used to demonstrate some simple uses of Observable.

HTTP Request Use Case

```
ObservableHttp.createRequest(
   HttpAsyncMethods.createGet("http://www.wikipedia.com"), client)
        .toObservable() // Observable<ObservableHttpResponse>
```

The request is lazy and we turn it into an Observable that when subscribed to will execute the request and callback with the response.

HTTP Request Use Case

```
ObservableHttp.createRequest(
   HttpAsyncMethods.createGet("http://www.wikipedia.com"), client)
        .toObservable() // Observable<ObservableHttpResponse>
        .flatMap((ObservableHttpResponse response) -> {
            // access to HTTP status, headers, etc
            // response.getContent() -> Observable<byte[]>
            return response.getContent().map((bb) -> {
                return new String(bb); // Observable<String>
            });
         })
```

Once we have the ObservableHttpResponse we can choose what to do with it, including fetching the content which returns an Observable<byte[]>.

HTTP Request Use Case

```
ObservableHttp.createRequest(
   HttpAsyncMethods.createGet("http://www.wikipedia.com"), client)
        .toObservable() // Observable<ObservableHttpResponse>
        .flatMap((ObservableHttpResponse response) -> {
            // access to HTTP status, headers, etc
            // response.getContent() -> Observable<byte[]>
            return response.getContent().map((bb) -> {
                return new String(bb); // Observable<String>
            });
        })
```

We use flatMap as we want to perform nested logic that returns another Observable, ultimately an Observable<String> in this example.

HTTP Request Use Case

```
ObservableHttp.createRequest(
   HttpAsyncMethods.createGet("http://www.wikipedia.com"), client)
        .toObservable() // Observable<ObservableHttpResponse>
        .flatMap((ObservableHttpResponse response) -> {
            // access to HTTP status, headers, etc
            // response.getContent() -> Observable<byte[]>
            return response.getContent().map((bb) -> {
                return new String(bb); // Observable<String>
            });
         })
```

We use map to transform from byte[] to String and return that.

HTTP Request Use Case

```
ObservableHttp.createRequest(
   HttpAsyncMethods.createGet("http://www.wikipedia.com"), client)
        .toObservable() // Observable<ObservableHttpResponse>
        .flatMap((ObservableHttpResponse response) -> {
            // access to HTTP status, headers, etc
            // response.getContent() -> Observable<byte[]>
            return response.getContent().map((bb) -> {
                return new String(bb); // Observable<String>
            });
         })
        .subscribe((resp) -> {
             System.out.println(resp);
        });
```

We can subscribe to this asynchronously ...

HTTP Request Use Case

```
ObservableHttp.createRequest(
   HttpAsyncMethods.createGet("http://www.wikipedia.com"), client)
        .toObservable() // Observable<ObservableHttpResponse>
        .flatMap((ObservableHttpResponse response) -> {
            // access to HTTP status, headers, etc
            // response.getContent() -> Observable<byte[]>
            return response.getContent().map((bb) -> {
                return new String(bb); // Observable<String>
            });
         })
        .subscribe((resp) -> {
             System.out.println(resp);
        });
```

... which will execute all of the lazily defined code above and receive String results.

HTTP Request Use Case

```
ObservableHttp.createRequest(
   HttpAsyncMethods.createGet("http://www.wikipedia.com"), client)
        .toObservable() // Observable<ObservableHttpResponse>
        .flatMap((ObservableHttpResponse response) -> {
            // access to HTTP status, headers, etc
            // response.getContent() -> Observable<byte[]>
            return response.getContent().map((bb) -> {
                return new String(bb); // Observable<String>
            });
         })
        .toBlockingObservable()
        .forEach((resp) -> {
             System.out.println(resp);
        });
```

Or if we need to be blocking (useful for unit tests or simple demo apps) we can use toBlockingObservable().forEach() to iterate the responses in a blocking manner.

HTTP Request Use Case

```
ObservableHttp.createGet("http://www.wikipedia.com"), client)
        .toObservable() // Observable<ObservableHttpResponse>
        .flatMap((ObservableHttpResponse response) -> {
            // access to HTTP status, headers, etc
            // response.getContent() -> Observable<byte[]>
            return response.getContent().map((bb) -> {
                return new String(bb); // Observable<String>
            });
         })
```

This example has shown just a simple request/response.

HTTP Request Use Case

```
ObservableHttp.createGet("http://www.wikipedia.com"), client)
        .toObservable() // Observable<ObservableHttpResponse>
        .flatMap((ObservableHttpResponse response) -> {
            // access to HTTP status, headers, etc
            // response.getContent() -> Observable<byte[]>
            return response.getContent().map((bb) -> {
                return new String(bb); // Observable<String>
            });
         })
```

If we change the request ...

HTTP Request Use Case

```
ObservableHttp.createGet("http://hostname/hystrix.stream"), client)
        .toObservable() // Observable<ObservableHttpResponse>
        .flatMap((ObservableHttpResponse response) -> {
            // access to HTTP status, headers, etc
            // response.getContent() -> Observable<byte[]>
            return response.getContent().map((bb) -> {
                return new String(bb); // Observable<String>
            });
         })
```

... to something that streams results (mime-type text/event-stream) we can see a more interesting use of Observable.

HTTP Request Use Case

```
ObservableHttp.createGet("http://hostname/hystrix.stream"), client)
        .toObservable() // Observable<ObservableHttpResponse>
        .flatMap((ObservableHttpResponse response) -> {
            // access to HTTP status, headers, etc
            // response.getContent() -> Observable<byte[]>
            return response.getContent().map((bb) -> {
                return new String(bb); // Observable<String>
            });
         })
         .filter((s) -> {
            s.startsWith(": ping");
         })
         .take(30);
```

We will receive a stream (potentially infinite) of events.

HTTP Request Use Case

```
ObservableHttp.createGet("http://hostname/hystrix.stream"), client)
        .toObservable() // Observable<ObservableHttpResponse>
        .flatMap((ObservableHttpResponse response) -> {
            // access to HTTP status, headers, etc
            // response.getContent() -> Observable<byte[]>
            return response.getContent().map((bb) -> {
                return new String(bb); // Observable<String>
            });
        })
        .filter((s) -> {
            s.startsWith(": ping");
        })
        .take(30);
```

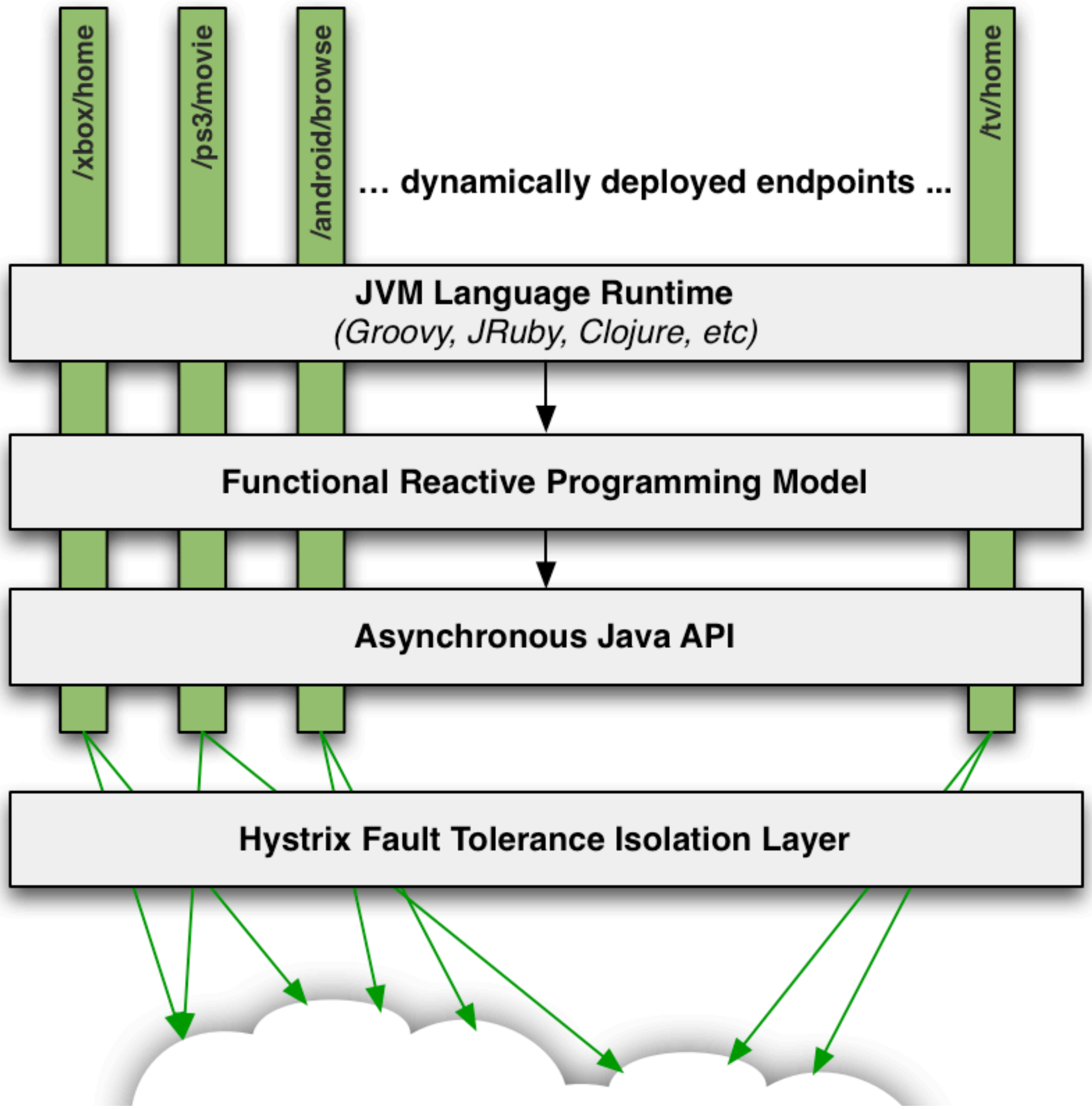
We can filter out all ": ping" events ...

HTTP Request Use Case

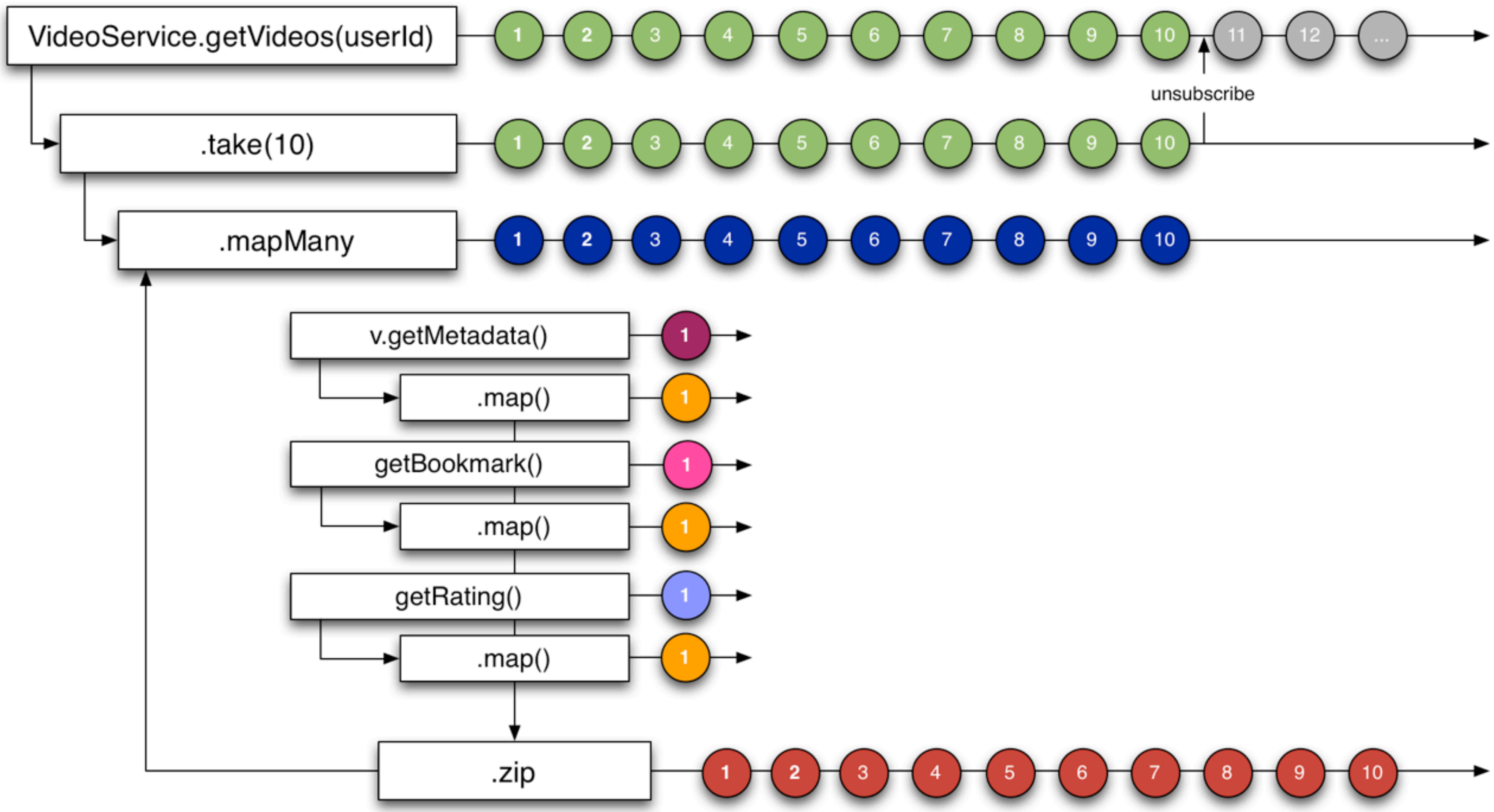
```
ObservableHttp.createGet("http://hostname/hystrix.stream"), client)
        .toObservable() // Observable<ObservableHttpResponse>
        .flatMap((ObservableHttpResponse response) -> {
            // access to HTTP status, headers, etc
            // response.getContent() -> Observable<byte[]>
            return response.getContent().map((bb) -> {
                return new String(bb); // Observable<String>
            });
         })
         .filter((s) -> {
            s.startsWith(": ping");
         })
         .take(30);
```

... and take the first 30 and then unsubscribe. Or we can use operations like window/buffer/groupBy/scan to group and analyze the events.

Netflix API Use Case

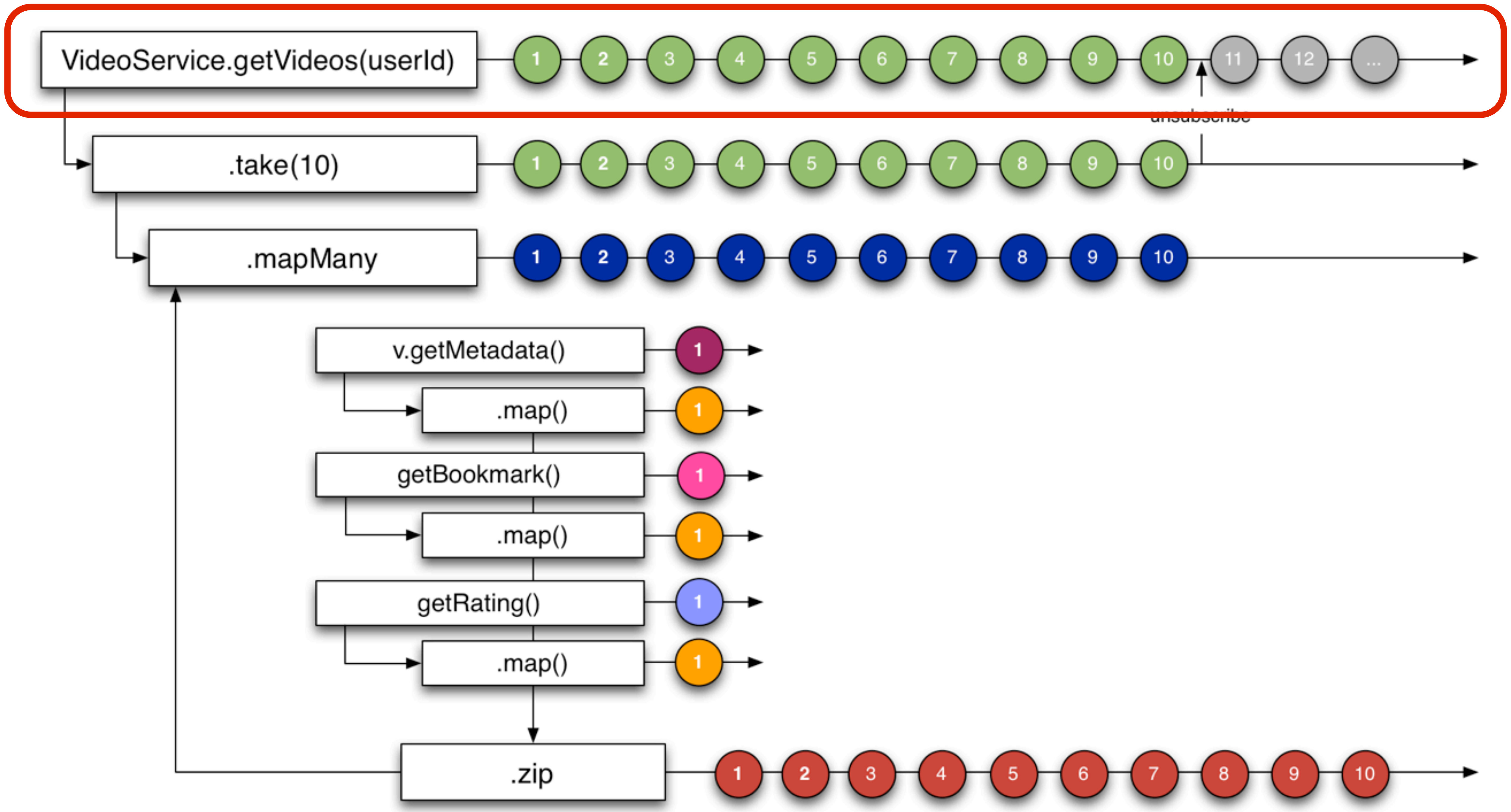


Now we'll move to a more involved example of how Rx is used in the Netflix API that demonstrates some of the power of Rx to handle nested asynchronous composition.



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

This marble diagram represents what the code in subsequent slides is doing when retrieving data and composing the functions.



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

Observable<Video> emits n videos to onNext()

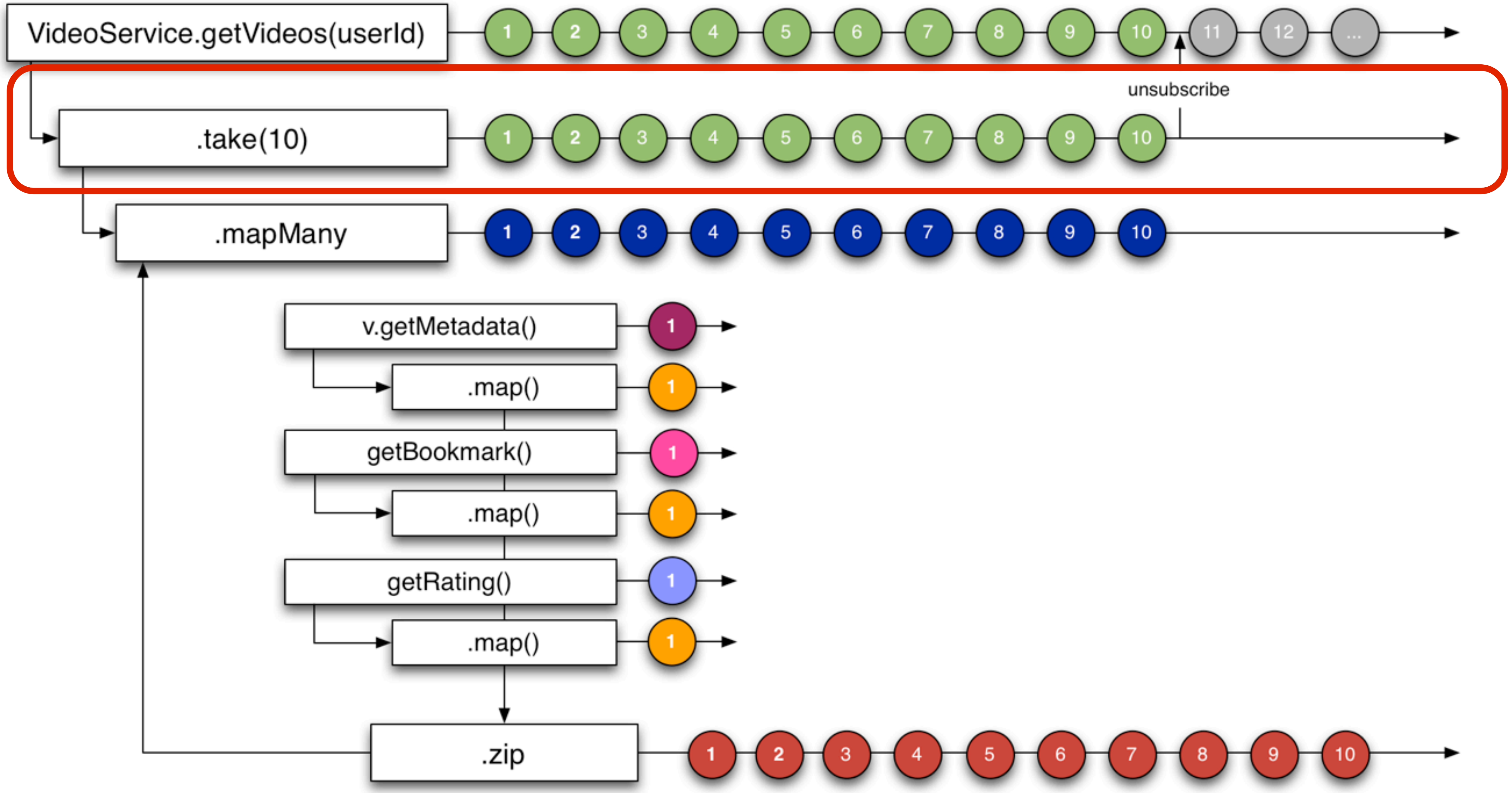
First we start with a request to fetch videos asynchronously ...

```
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)
}
```

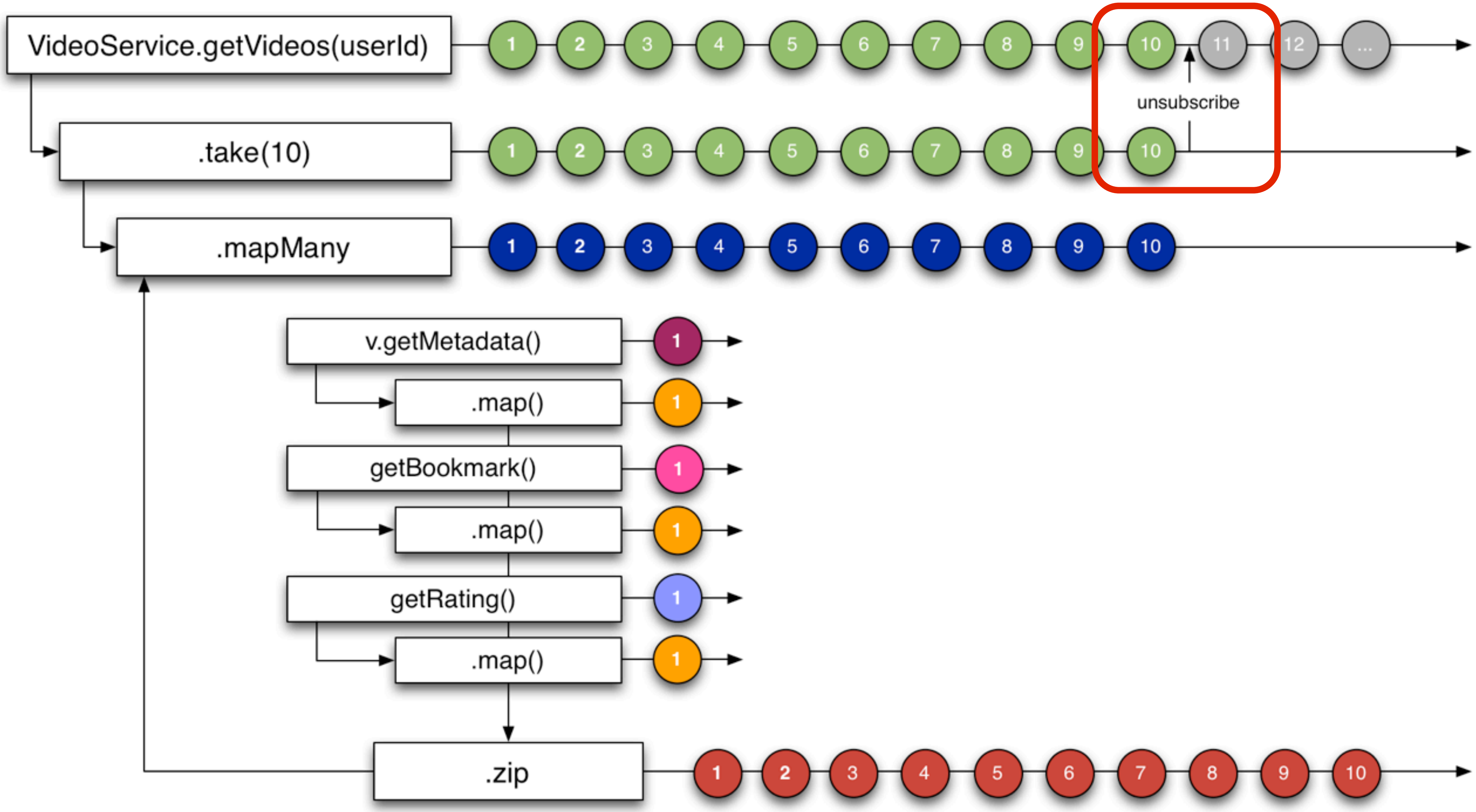
Takes first 10 then unsubscribes from origin.
Returns Observable<Video> that emits 10 Videos.



[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.

The take operator subscribes to the Observable from VideoService.getVideos, accepts 10 onNext calls ...



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

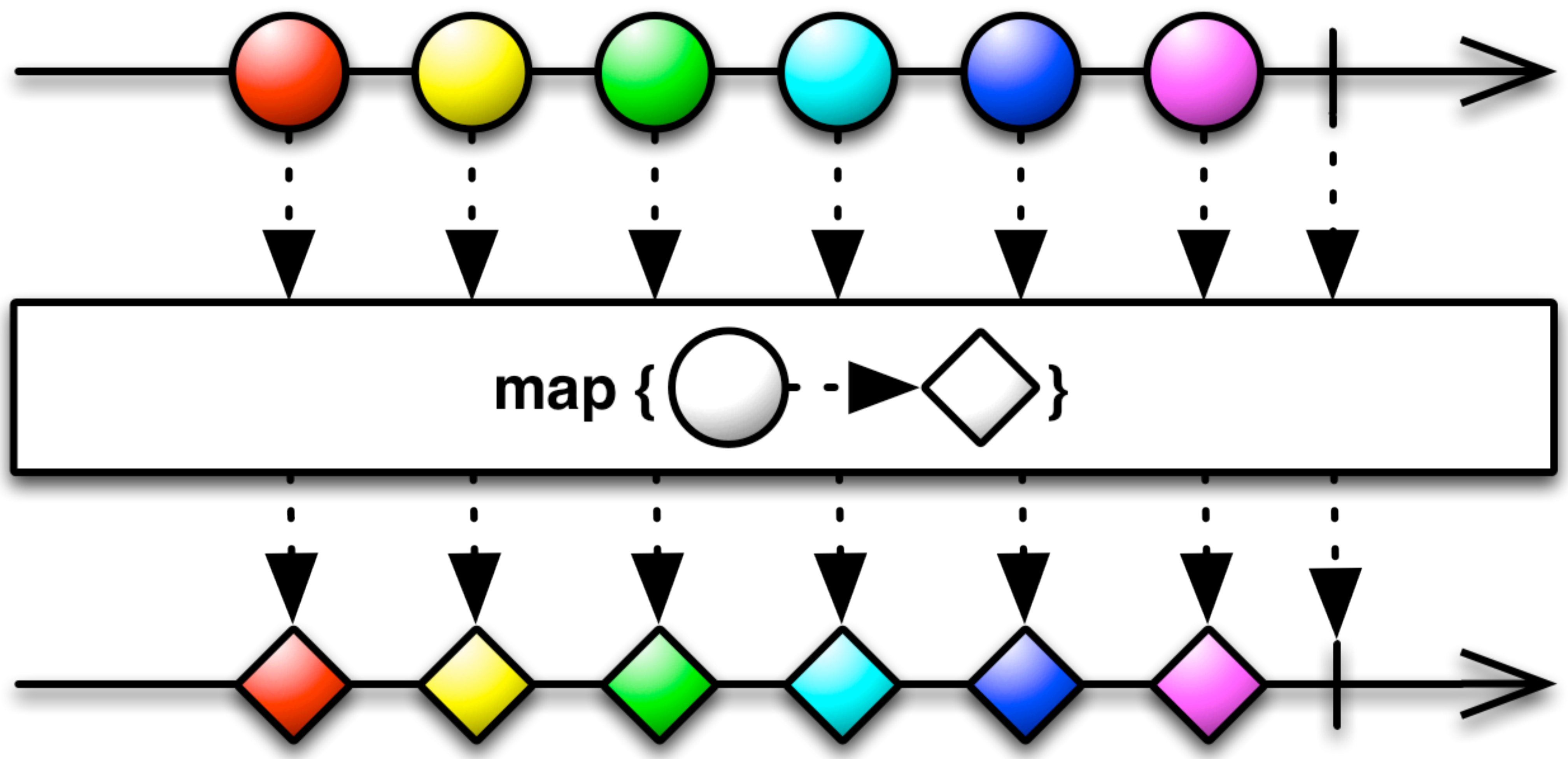
Takes first 10 then unsubscribes from origin.
Returns Observable<Video> that emits 10 Videos.

... and then unsubscribes from the parent Observable so only 10 Video objects are emitted from the 'take' Observable. The parent Observable receives the unsubscribe call and can stop further processing, or if it incorrectly ignores the unsubscribe the 'take' operator will ignore any further data it receives.

```
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
        })
}
```

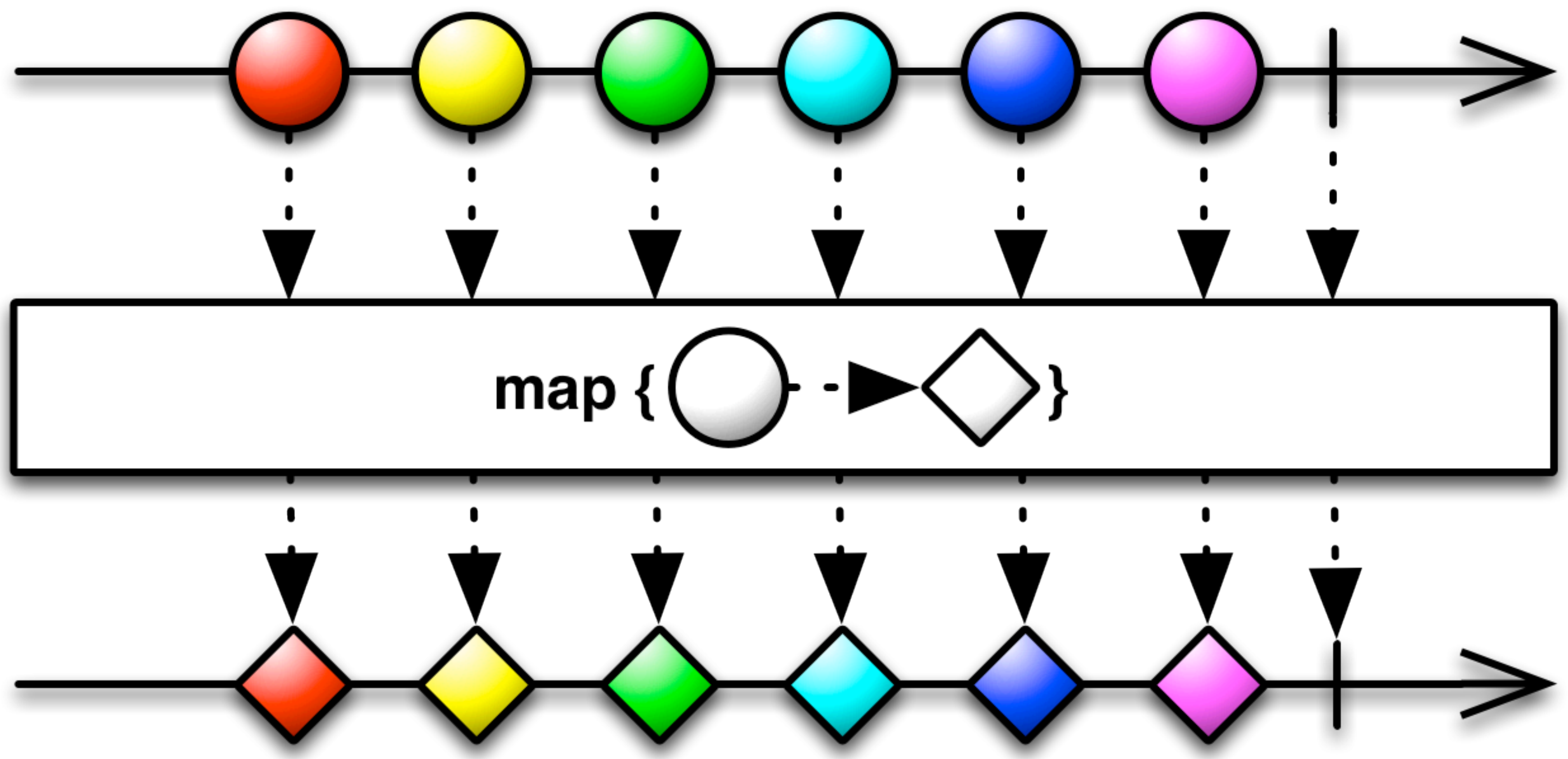
The 'map' operator allows transforming the input value into a different output.

We now apply the 'map' operator to each of the 10 Video objects we will receive so we can transform from Video to something else.




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

The ‘map’ operator allows transforming from type T to type R.




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

The ‘map’ operator allows transforming from type T to type R.

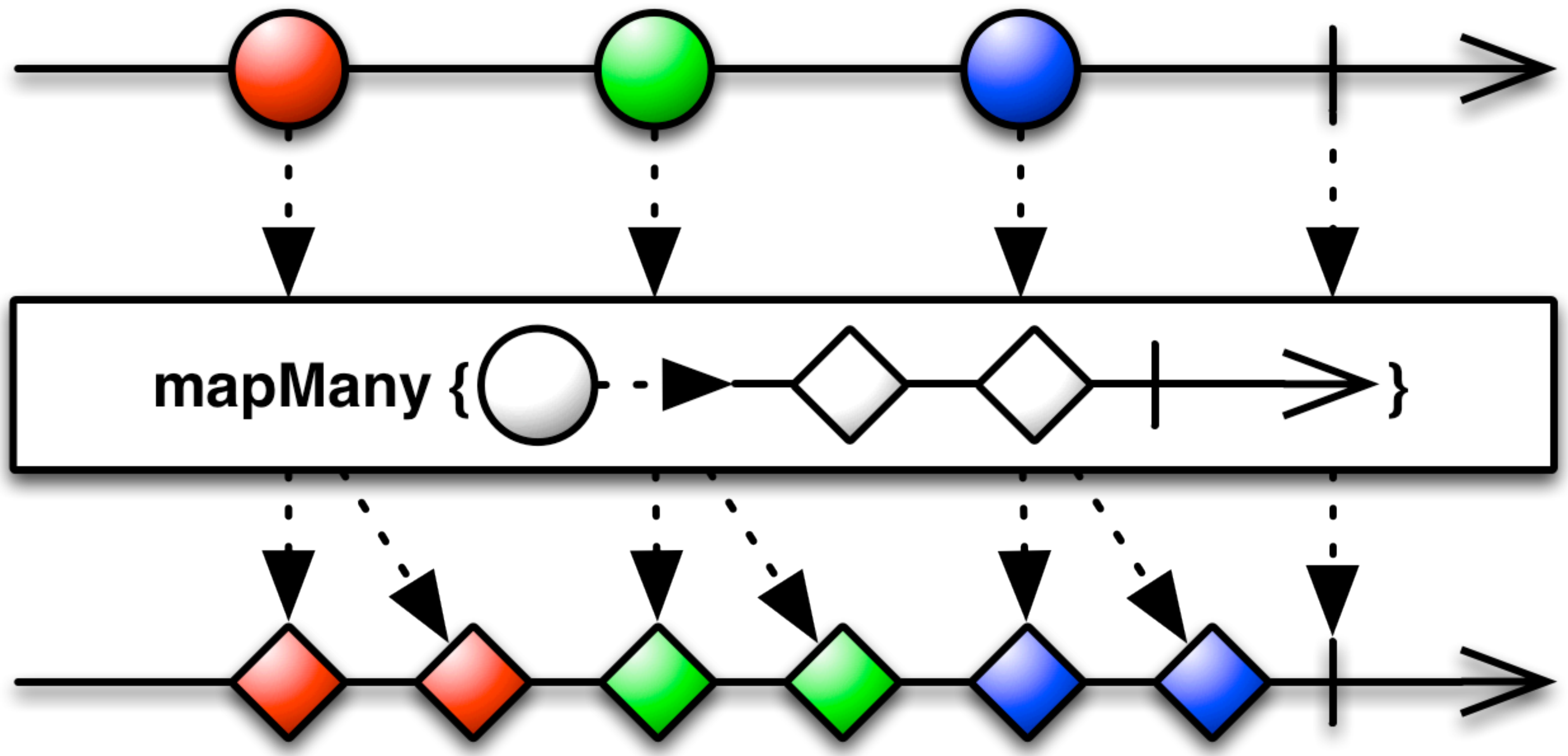
```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .map({ Video video ->
           // transform video object
        })
}
```

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.

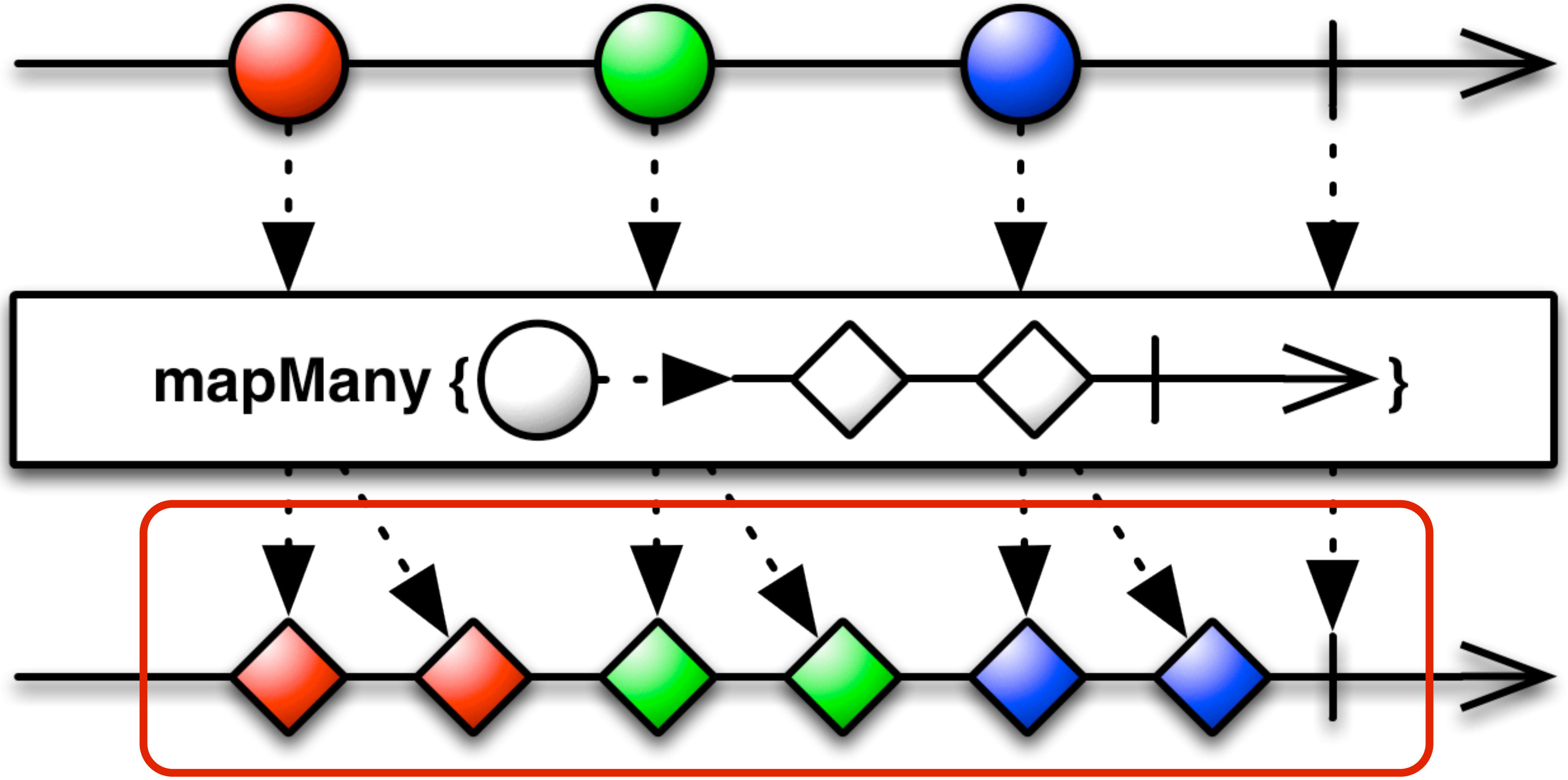
But since we want to do nested asynchronous calls that will result in another Observable being returned we will use flatMap (also knows as mapMany or selectMany) which will flatten an Observable<Observable<T>> into Observable<T> as shown in the following marble diagram ...




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

The ‘flatMap’/‘mapMany’ operator allows transforming from type T to type Observable<R>. If ‘map’ were being used this would result in an Observable<Observable<R>> which is rarely what is wanted, so ‘flatMap’/‘mapMany’ flattens this via ‘merge’ back into Observable<R>.

This is generally used instead of ‘map’ anytime nested work is being done that involves fetching and returning other Observables.




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

A single flattened Observable<R> is returned instead of Observable<Observable<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.

Within the flatMap "transformation" function we perform 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.

This call returns an Observable<VideoMetadata>.

```
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>

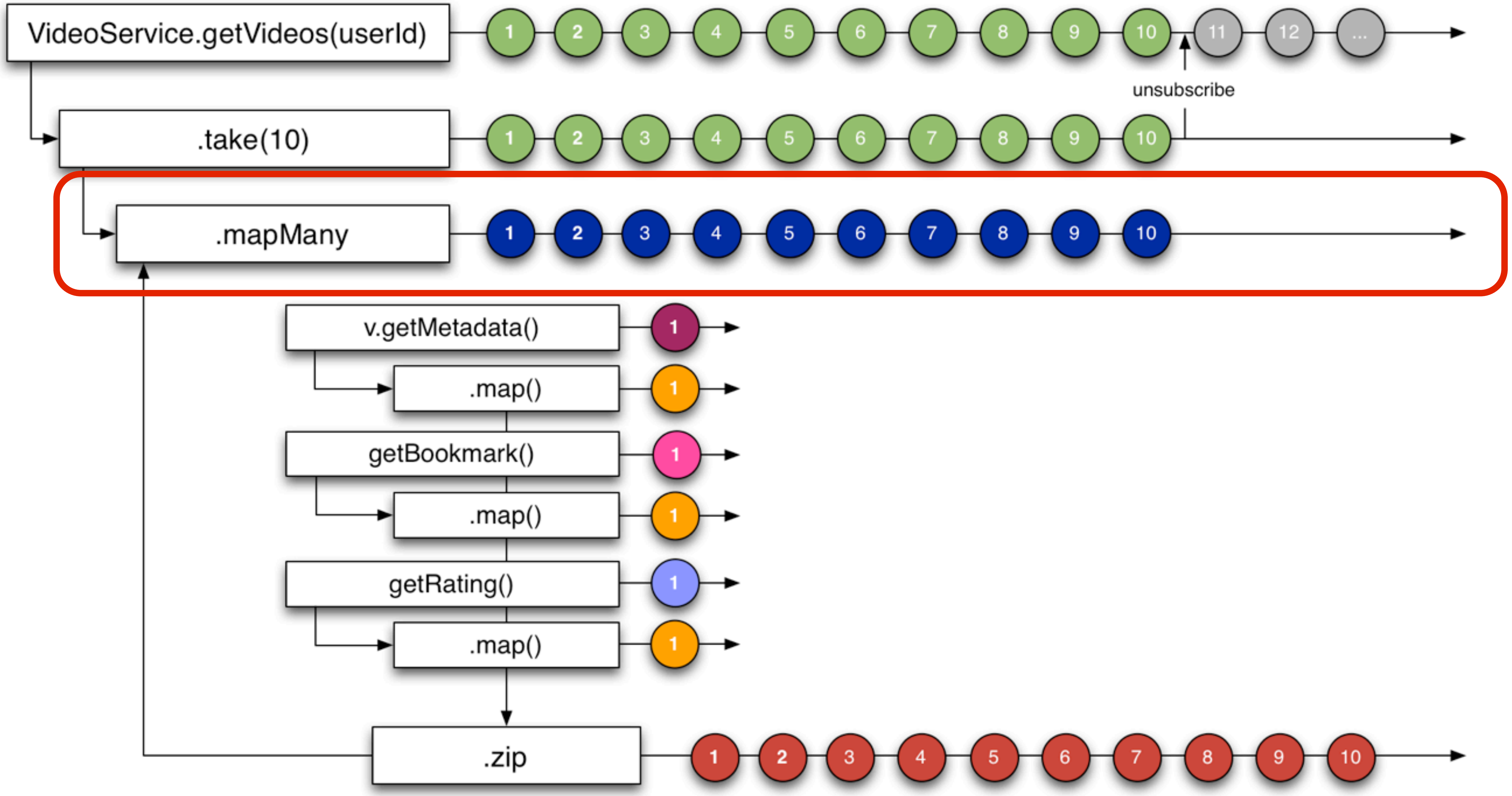
Observable<VideoRating>

3 separate types are being fetched asynchronously and each return an Observable.

```
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 ...
        })
}
```

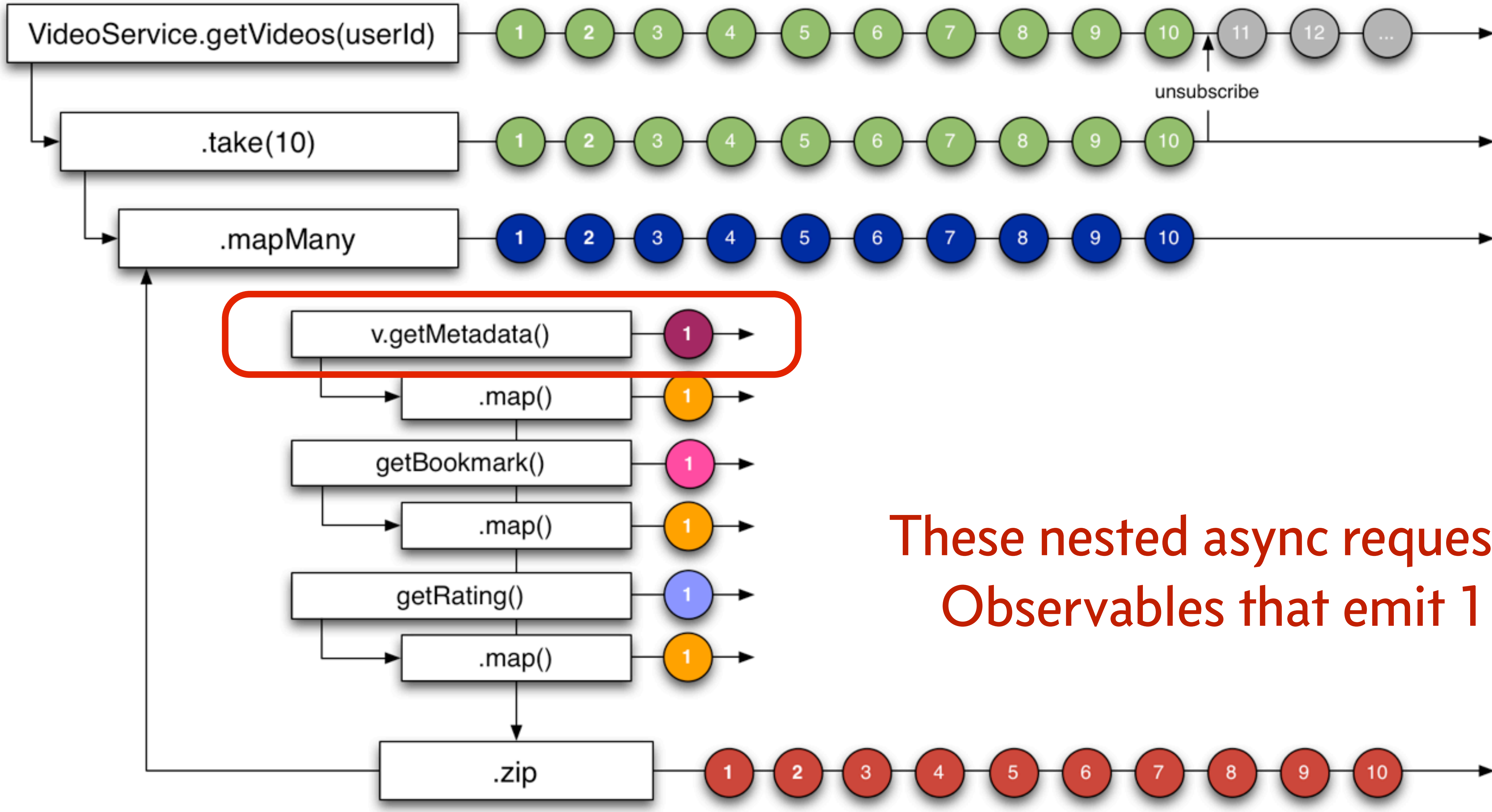
Each Observable transforms its data using 'map'

Each of the 3 different Observables are transformed using 'map', in this case from the VideoMetadata type into a dictionary of key/value pairs.



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

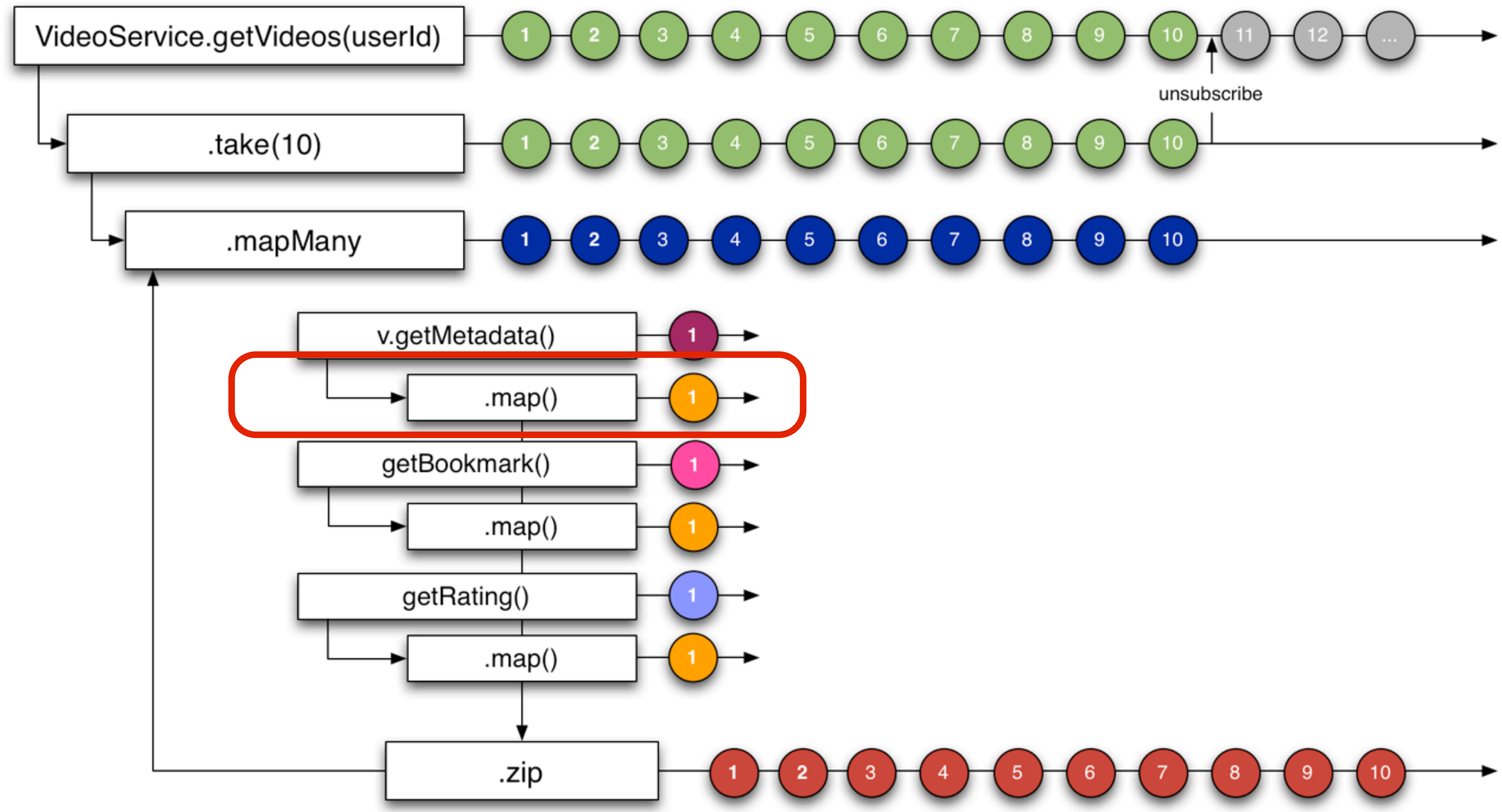
For each of the 10 Video objects it transforms
via 'mapMany' function that does nested async calls.



These nested async requests return Observables that emit 1 value.

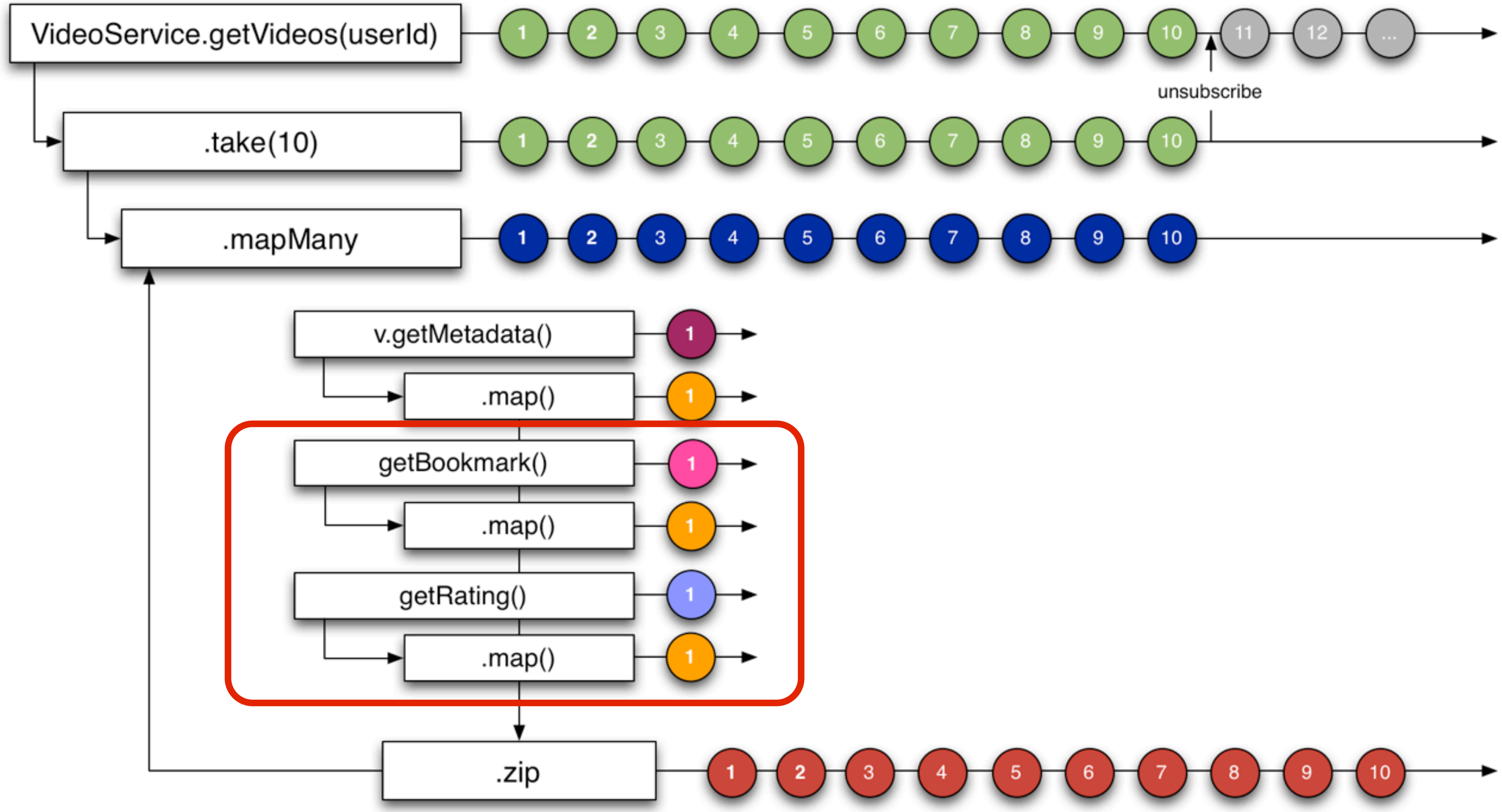
[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

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>

Each of the .map() calls emits the same type (represented as an orange circle) since we want to combine them later into a single dictionary (Map).

```
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
    })
}
```

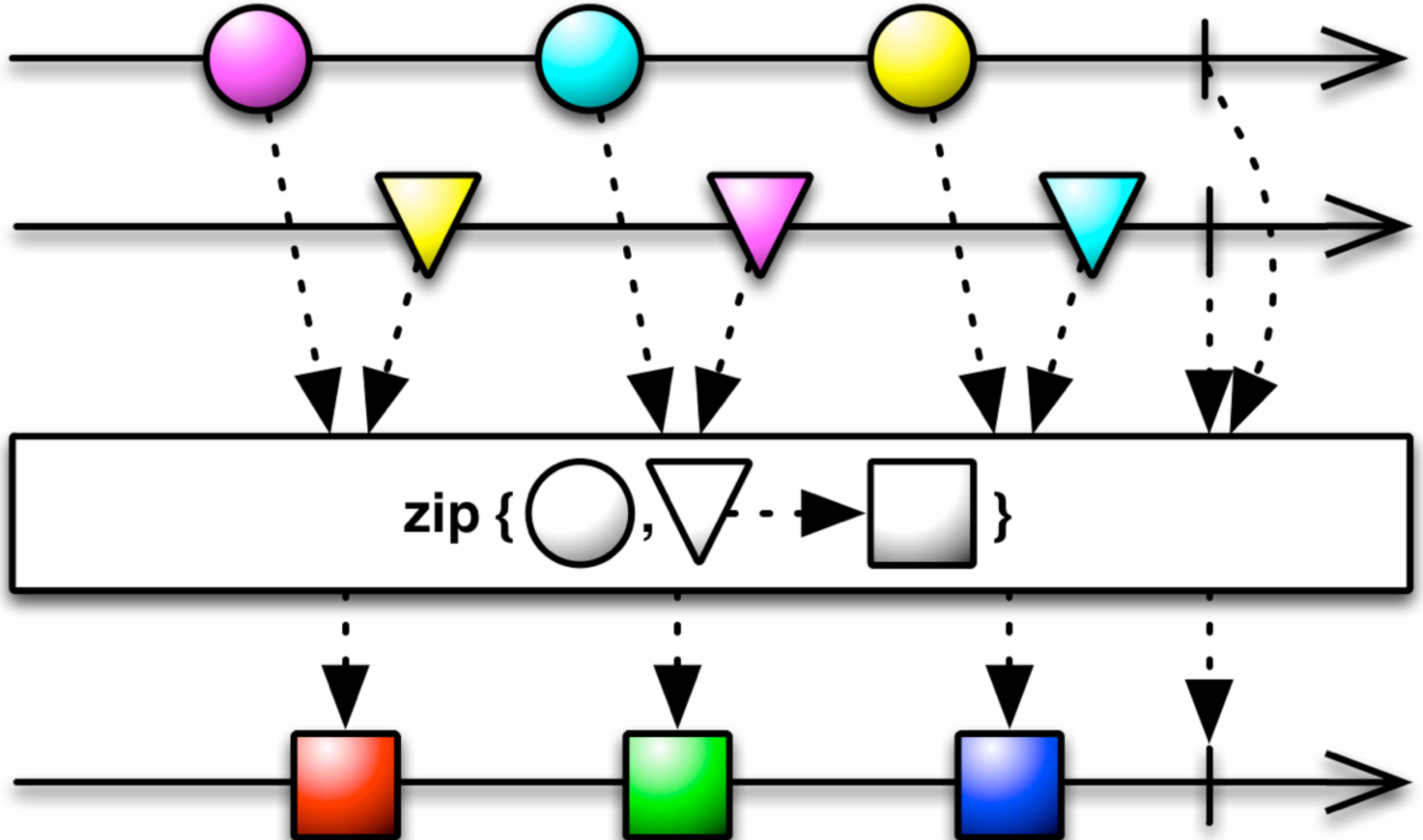
At this point we have 3 Observables defined but they are dangling - nothing combines or references them and we aren't yet returning anything from the 'flatMap' function so we want to compose m, b, and r together and return a single asynchronous Observable representing the composed work being done on those 3.

```
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
          })
    })
}
```

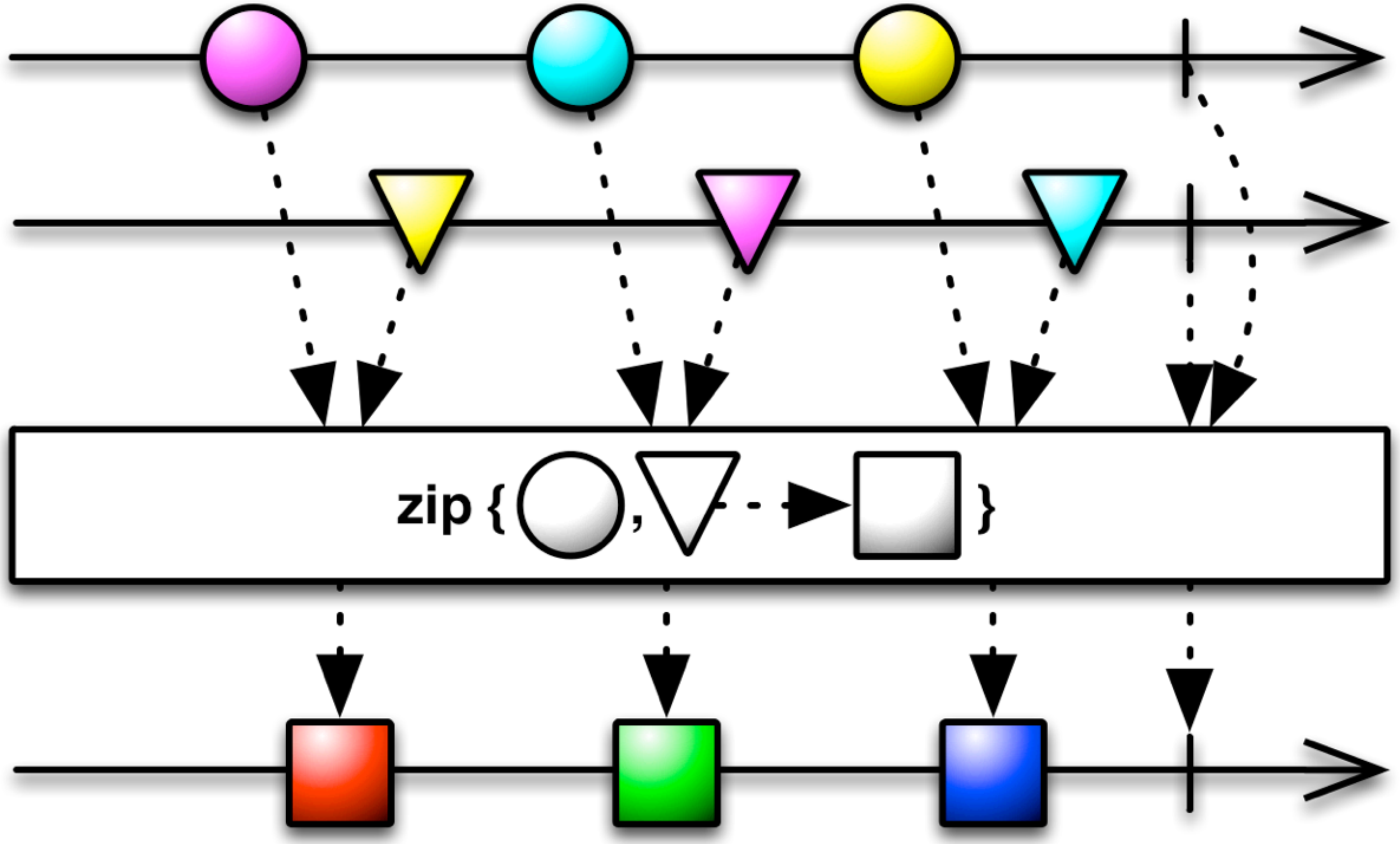
```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .flatMap({ Video video ->
            def m ...
            def b ...
            def r ...
            // compose these together
        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

We use ‘zip’ to combine the 3 together and apply a function to transform them into a single combined format that we want, in this case a dictionary that contains the key values pairs from the dictionaries emitted by ‘metadata’, ‘bookmark’, and ‘ratings’ along with the videoId also available within scope of the flatMap function and ‘closed over’ by the closure being executed in ‘zip’.




```
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
})
```

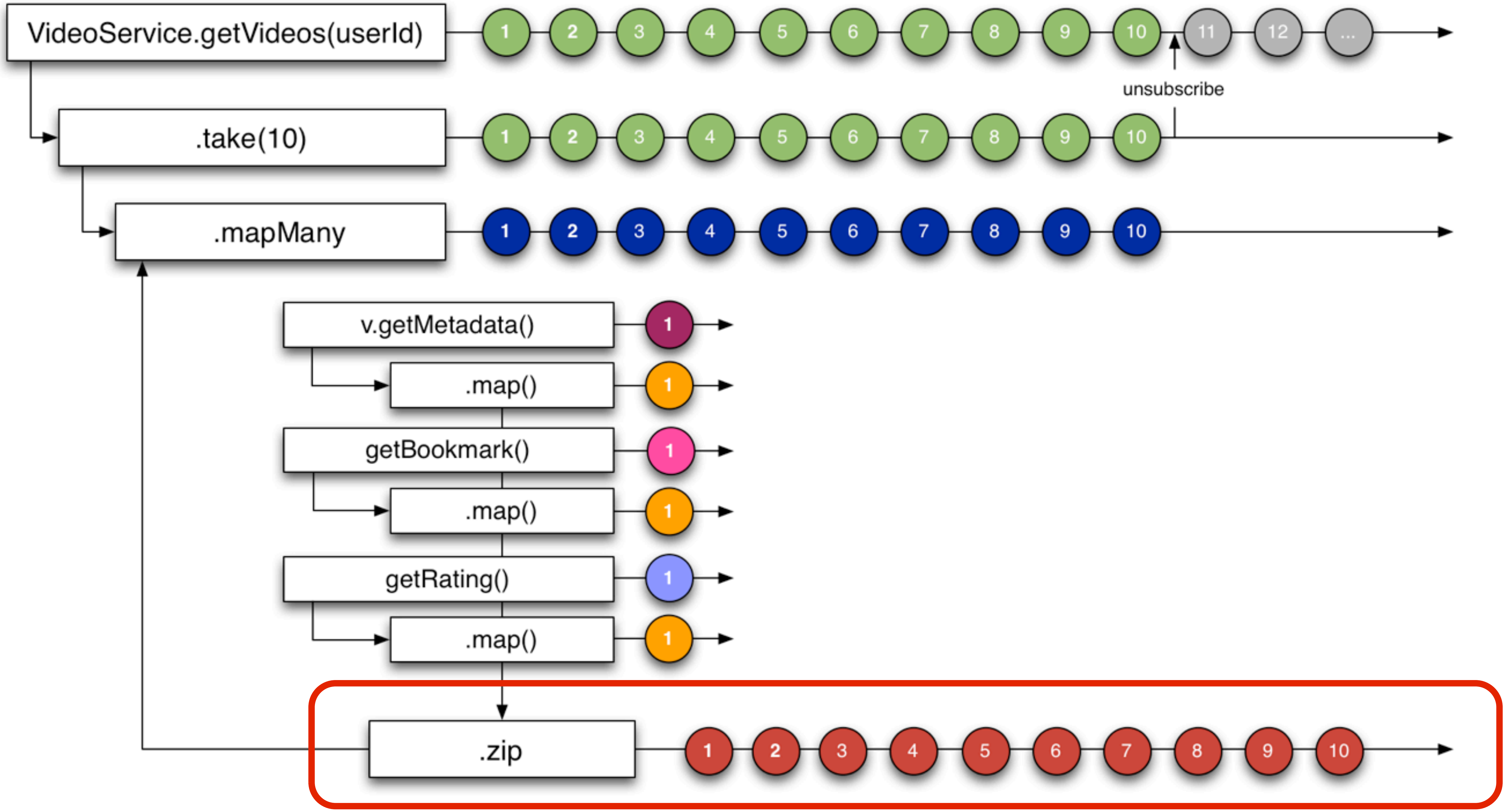
```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .flatMap({ Video video ->
          def m ...
          def b ...
          def r ...
          // compose these together
          return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

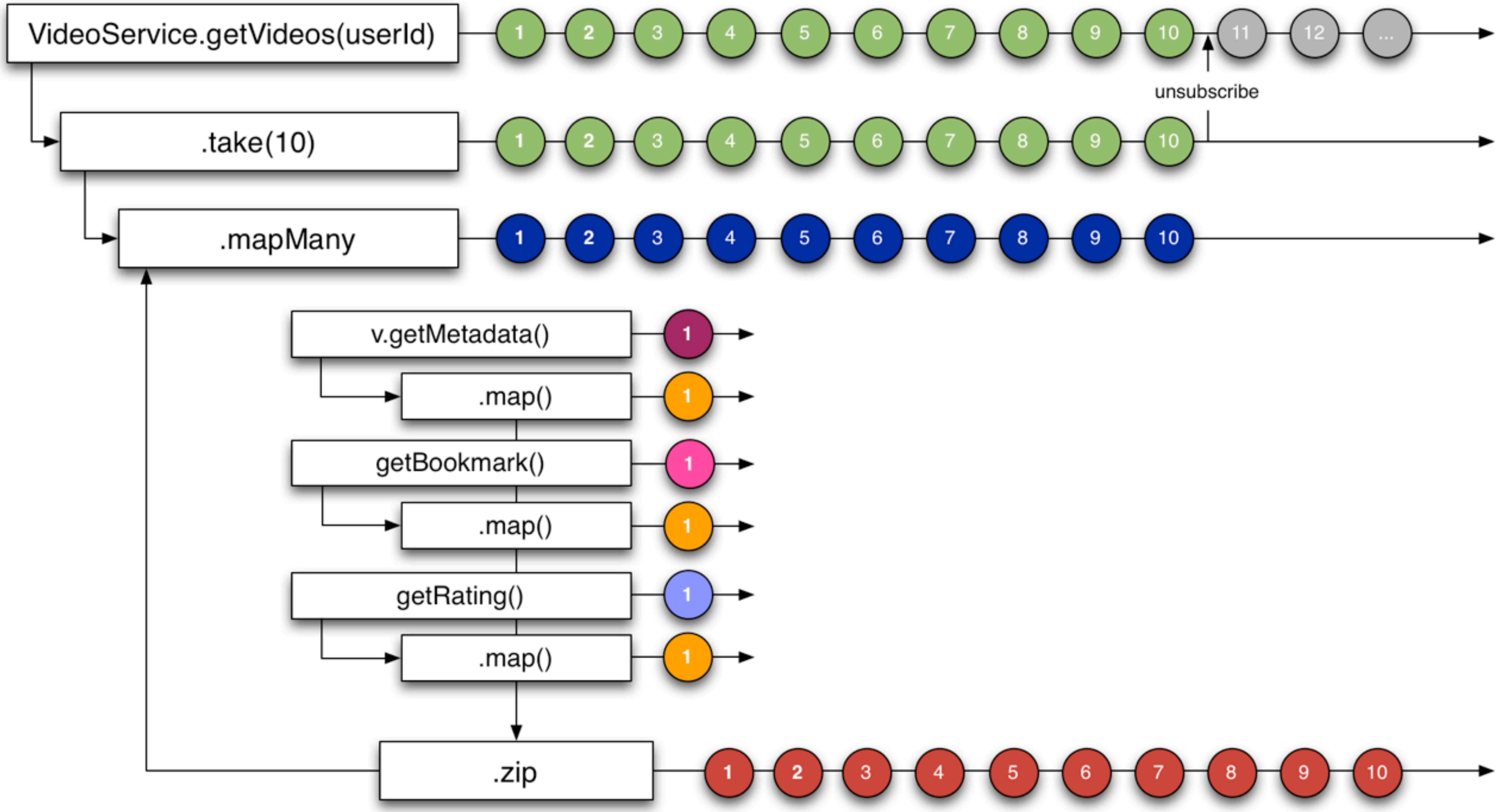
The entire composed Observable emits 10 Maps (dictionaries) of key/value pairs for each of the 10 VIdeo objects it receives.



[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.

The entire composed Observable emits 10 Maps (dictionaries) of key/value pairs for each of the 10 VIdeo objects it receives.



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

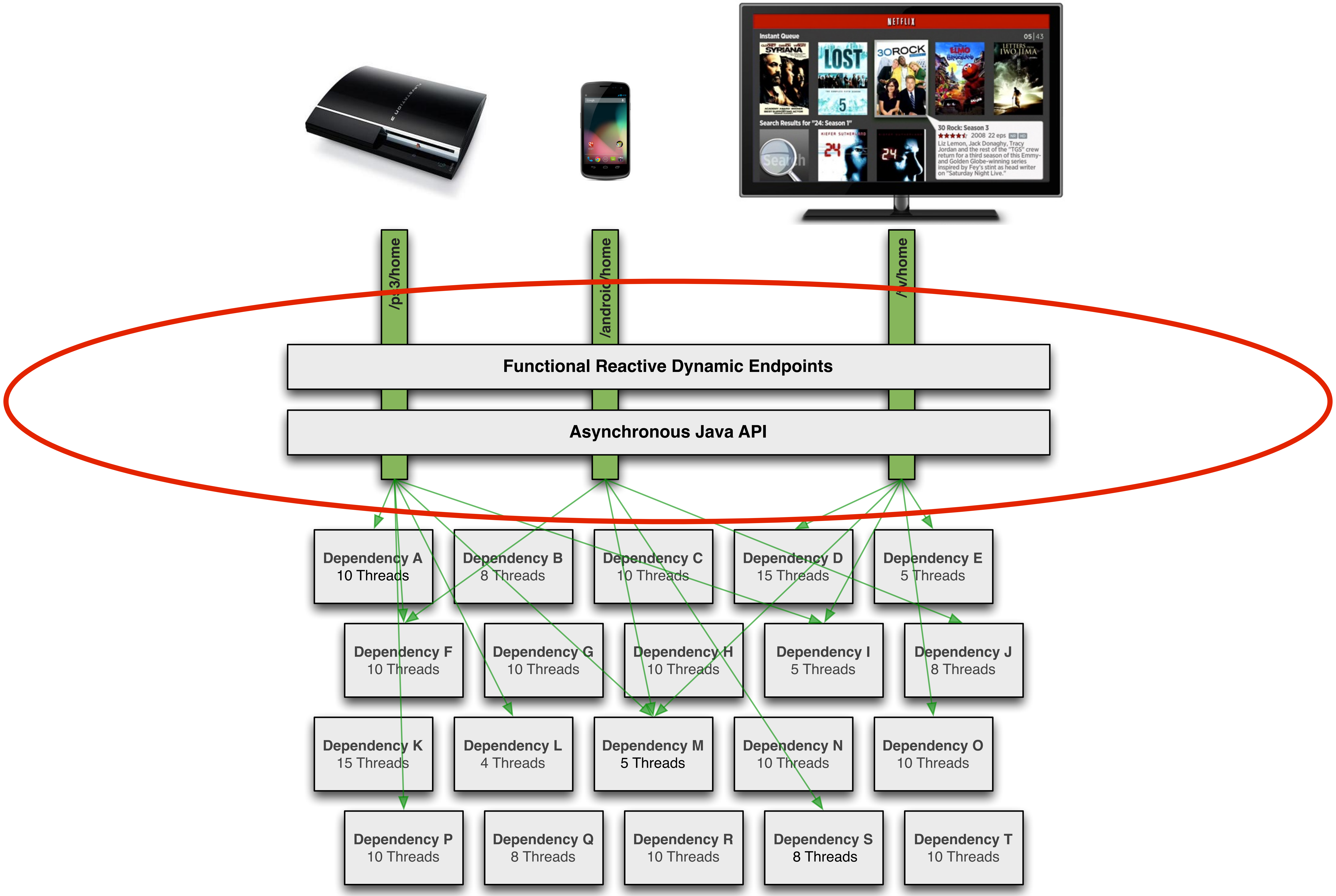
The full sequence returns Observable<Map> that emits a Map (dictionary) for each of 10 Videos.

INTERACTIONS WITH THE API

ARE ASYNCHRONOUS AND DECLARATIVE

API IMPLEMENTATION CONTROLS

CONCURRENCY BEHAVIOR



We have found Rx to be a good fit for creating Observable APIs and composing asynchronous data together while building web services using this approach.

With the success of Rx at the top layer of our stack we're now finding other areas where we want this programming model applied.

+

```
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()]
})
```

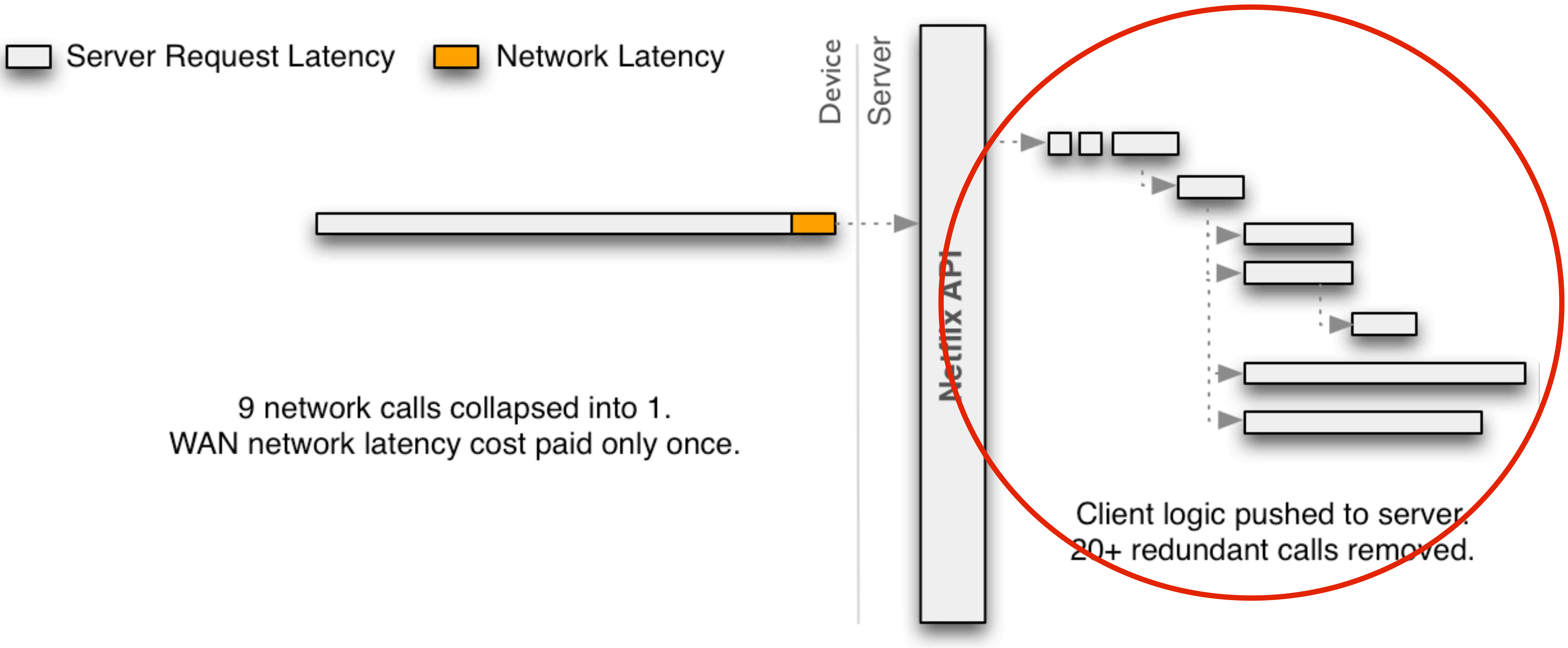
RxJava in Hystrix 1.3+

https://github.com/Netflix/Hystrix

One example of us pushing Rx deeper into our stack is the addition of support for RxJava to Hystrix version 1.3.

More information on the release can be found at https://github.com/Netflix/Hystrix/releases/tag/1.3.0

OBSERVABLE APIs



Looking back, Rx has enabled us to achieve our goals that started us down this path.

LESSONS LEARNED

Developer Training & Documentation

As we implemented and adopted Rx and enabled dozens of developers (most of them of either Javascript or imperative Java backgrounds) we found that workshops, training sessions and well-written documentation was very helpful in "onboarding" them to the new approach. We have found it generally takes a few weeks to get adjusted to the style.

LESSONS LEARNED

Developer Training & Documentation

Debugging and Tracing

Asynchronous code is challenging to debug. Improving our ability to debug, trace and visualize Rx “call graphs” is an area we are exploring.

LESSONS LEARNED

Developer Training & Documentation

Debugging and Tracing

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

Generally the model has been self-governing (get the code working and all is fine) but there has been one principle to teach since we are using this approach in mutable, imperative languages - don't mutate state outside the lambda/closure/function.

ASYNCHRONOUS

VALUES

EVENTS

PUSH

FUNCTIONAL REACTIVE

LAMBDAS

CLOSURES

(MOSTLY) PURE

COMPOSABLE

The Rx “functional reactive” approach is a powerful and straight-forward abstraction for asynchronously composing values and events and has worked well for the Netflix API.

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