Reactive Programing with RxJava

Piya Lumyong

Why?

Blocking
it’s Evil

we need

[illegible]

asynchronous code

but

How ?

Callbacks?

no composition

Callback Hell

Futures<T>?

to easy to block (get)

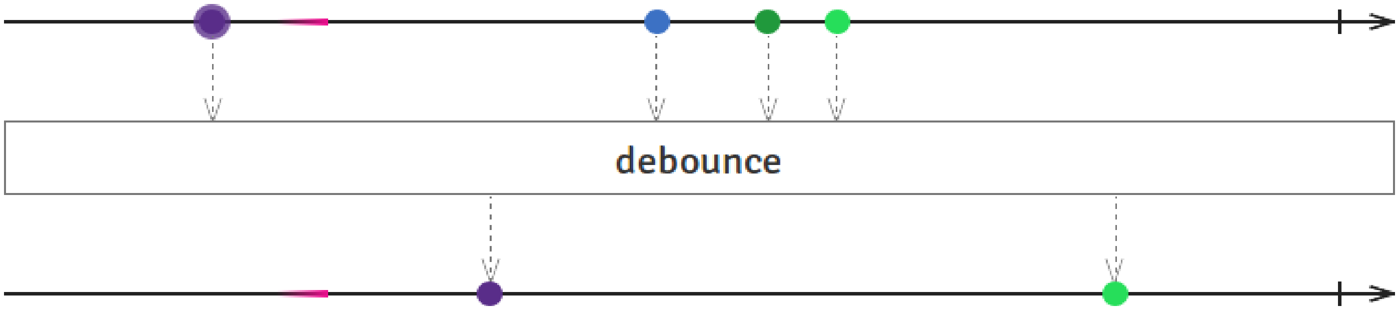
complex beyond 1 level
of composition

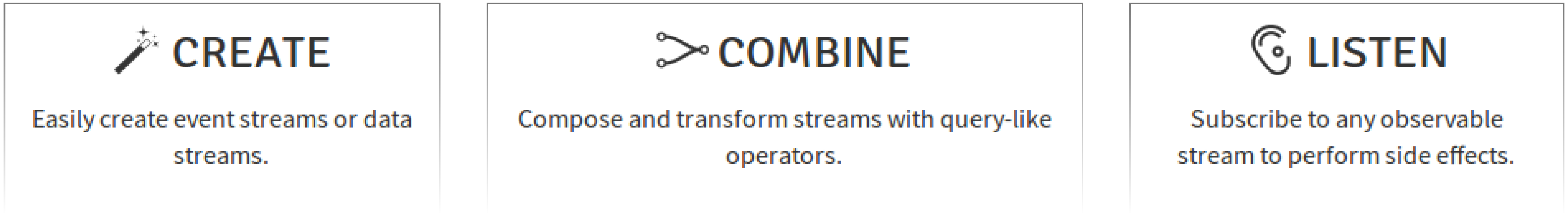
ReactiveX

An API for asynchronous programming
with observable streams

The Observer pattern done right

ReactiveX is a combination of the best ideas from
the Observer pattern, the Iterator pattern, and functional programming





CREATE

Easily create event streams or data streams.

COMBINE

Compose and transform streams with query-like operators.

LISTEN

Subscribe to any observable stream to perform side effects.

dual of

Iterable - Iterator “Pull”

become

Observable - Observer “Push”

ReactiveX extends the observer pattern for Async Programing

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

Iterable vs. Observable

event	Iterable(pull)	Observable(push)
retrieve data	T next()	onNext(T)
discover error	throw Exception()	onError(Exception)
complete	!hasNext()	onCompleted()

Gang of Four's Observer pattern missing

- The ability for the producer to signal to the consumer that there is no more data available.
- The ability for the producer to signal to the consumer that an error has occurred.

We use ReactiveX

SPRIN3r
Applied Duality
Microsoft
NETFLIX
GitHub
SOUNDCLOUD
Trello
treehouse
SeatGeek
ooVoo
Couchbase
futurice
welbe
airbnb

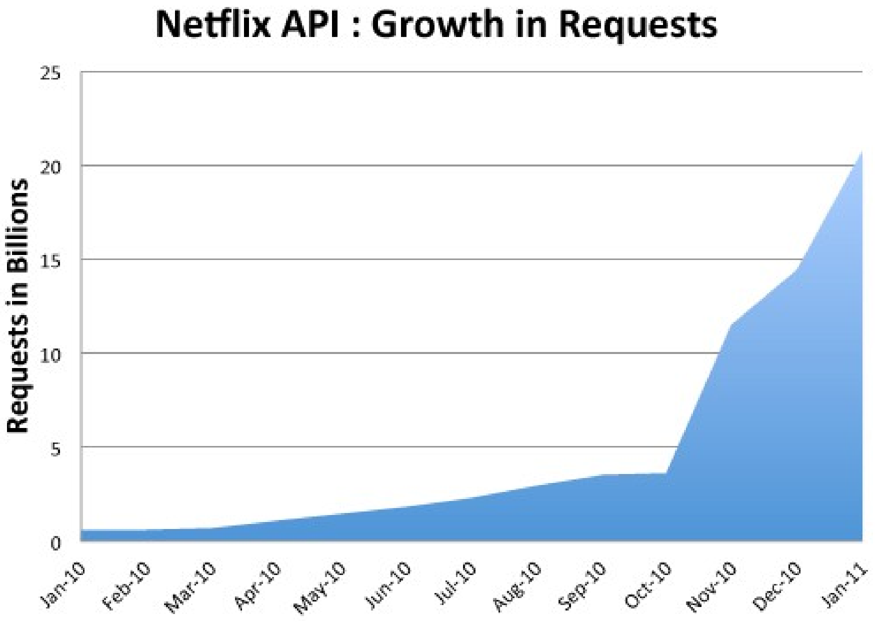
RxJava :

is a Java VM implementation of Reactive Extensions (ReactiveX)

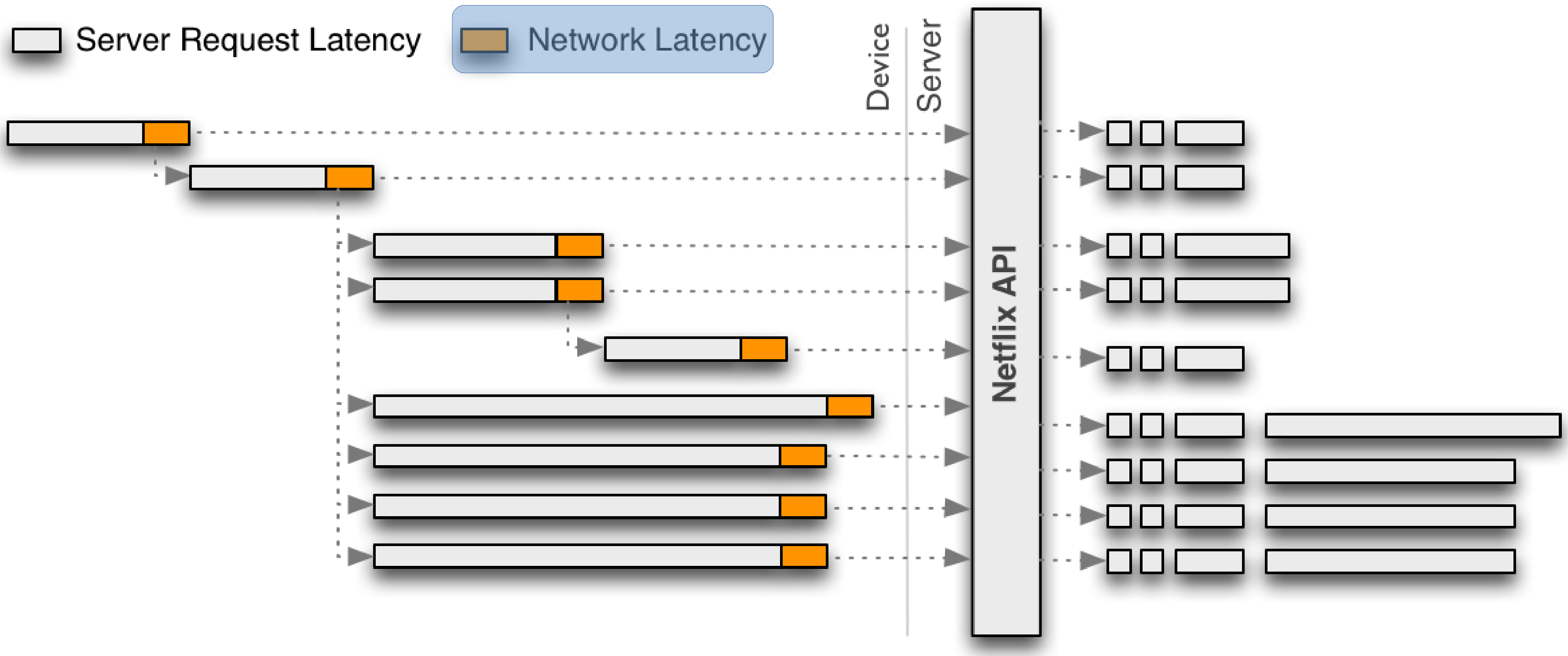
RxJava By NETFLIX

- Open source project with Apache License V2
- The Netflix API uses it to make the entire service layer asynchronous
- Provides a DSL for creating computation flows out of asynchronous sources using collection of operator for filtering, selecting, transforming and combining the flows in a lazy manner
- Targets the JVM not a language. Currently supports Java, Scala, Groovy, Clojure and Kotlin

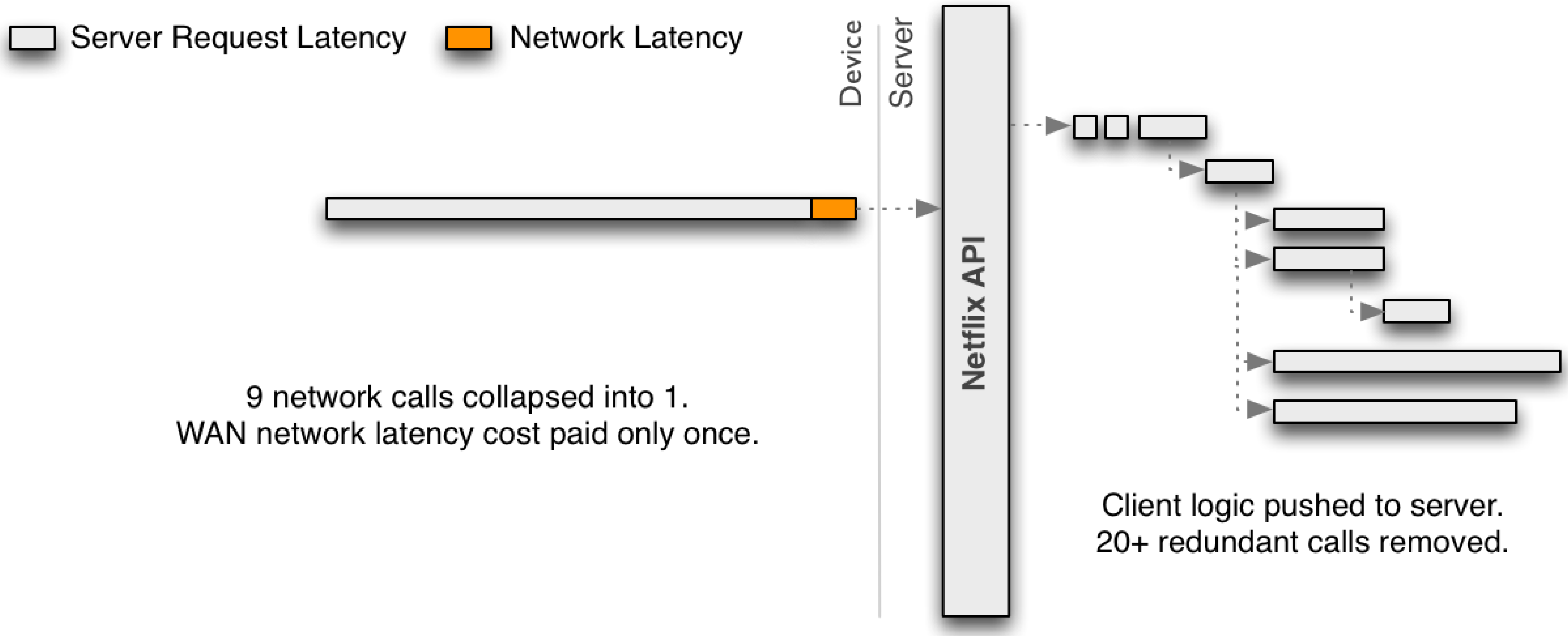
Redesigning the Netflix API



Client/Server communication



Reduce Chattiness



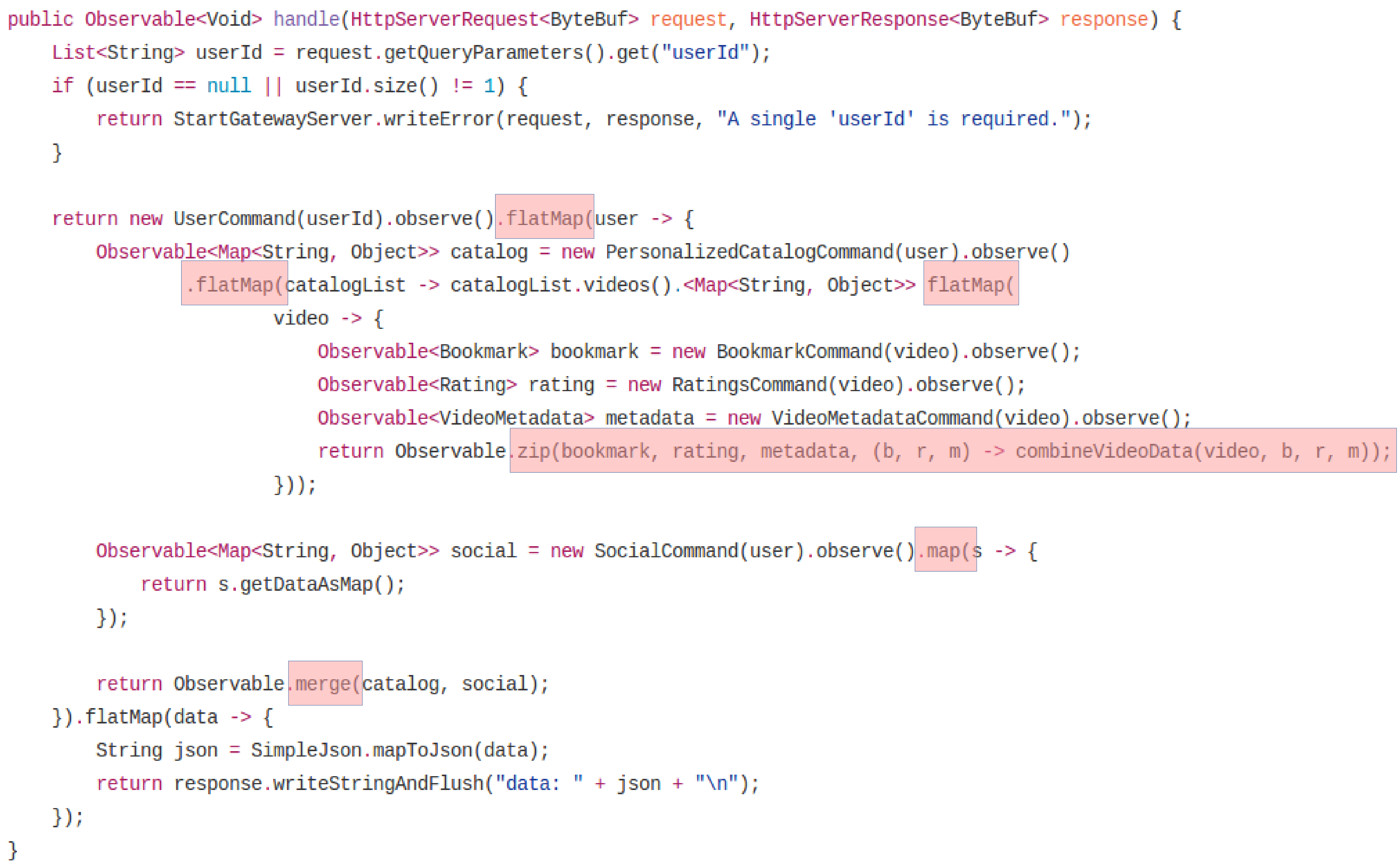
9 network calls collapsed into 1.
WAN network latency cost paid only once.

Client logic pushed to server.
20+ redundant calls removed.

```
/**
 * Demonstrate how Rx is used to compose Observables together
 * such as how a web service would to generate a JSON response.
 *
 * The simulated methods for the metadata represent different
 * services that are often backed by network calls.
 *
 * This will return a sequence of dictionaries such as this:
 *
 *  [id:1000, title:video-1000-title, length:5428, bookmark:0,
 *    rating:[actual:4, average:3, predicted:0]]
 */
def Observable getVideoGridForDisplay(userId) {
  getListOfLists(userId).mapMany({ VideoList list ->
    // for each VideoList we want to fetch the videos
    list.getVideos()
      .take(10) // we only want the first 10 of each list
      .mapMany({ 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")]
        })
        def b = video.getBookmark(userId).map({
          position ->
          return [bookmark: position]
        })
        def r = video.getRating(userId).map({
          VideoRating rating ->
          return [rating:
            [actual: rating.getActualStarRating(),
             average: rating.getAverageStarRating(),
             predicted: rating.getPredictedStarRating()]]
        })
        // 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
        })
      })
  })
}
```

```
// emits results such as
[id:1002, title:video-1002-title, length:5428, bookmark:0,
                     rating:[actual:2, average:4, predicted:3]]
[id:1003, title:video-1003-title, length:5428, bookmark:0,
                     rating:[actual:4, average:4, predicted:4]]
[id:1004, title:video-1004-title, length:5428, bookmark:0,
                     rating:[actual:4, average:1, predicted:1]]
```

RouteForDeviceHome



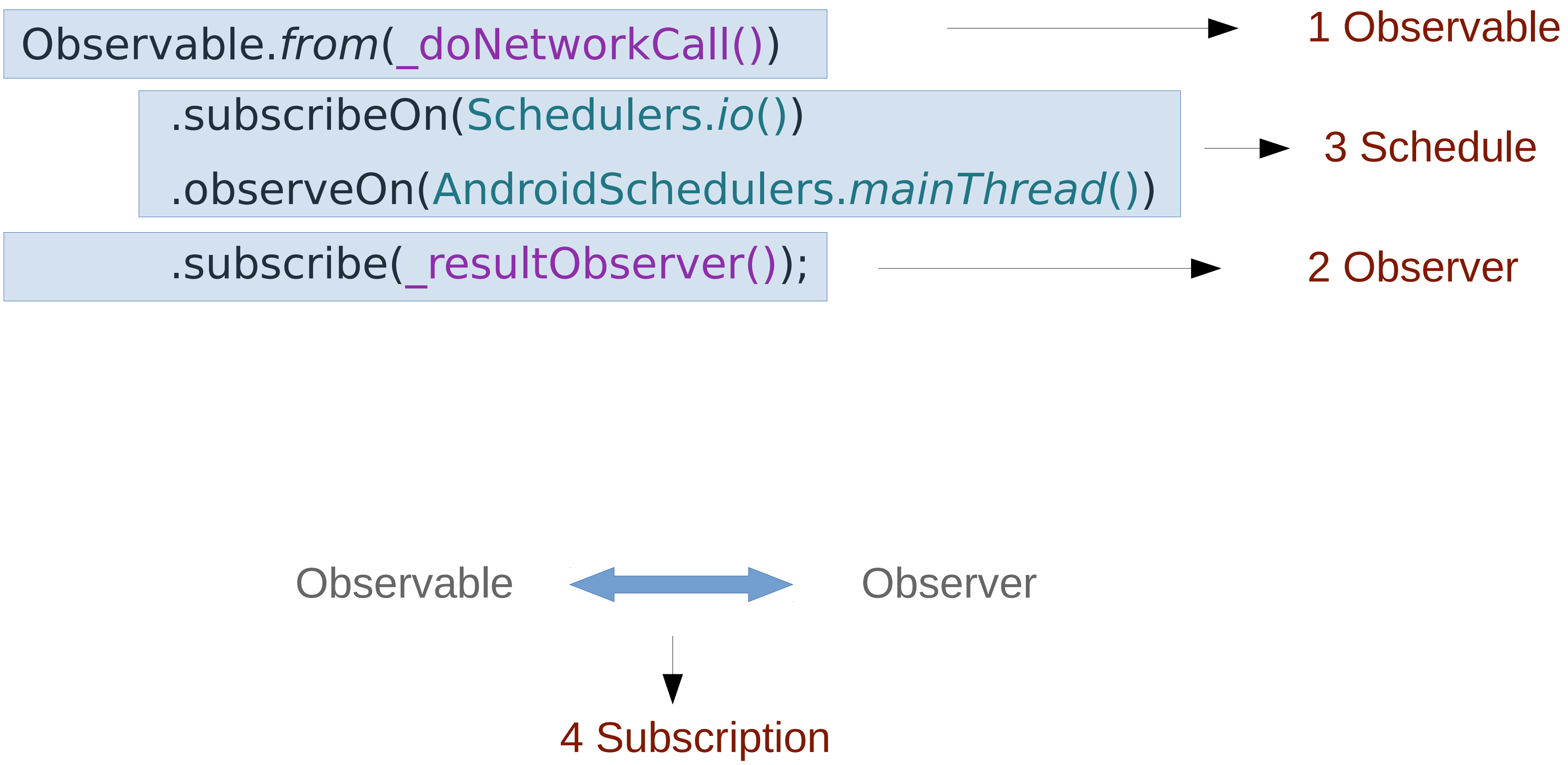
```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
    List<String> userId = request.getQueryParameters().get("userId");
    if (userId == null || userId.size() != 1) {
        return StartGatewayServer.writeError(request, response, "A single 'userId' is required.");
    }

    return new UserCommand(userId).observe().flatMap(user -> {
        Observable<Map<String, Object>> catalog = new PersonalizedCatalogCommand(user).observe()
                .flatMap(catalogList -> catalogList.videos().<Map<String, Object>> flatMap(
                        video -> {
                            Observable<Bookmark> bookmark = new BookmarkCommand(video).observe();
                            Observable<Rating> rating = new RatingsCommand(video).observe();
                            Observable<VideoMetadata> metadata = new VideoMetadataCommand(video).observe();
                            return Observable.zip(bookmark, rating, metadata, (b, r, m) -> combineVideoData(video, b, r, m));
                        }));

        Observable<Map<String, Object>> social = new SocialCommand(user).observe().map(s -> {
            return s.getDataAsMap();
        });

        return Observable.merge(catalog, social);
    }).flatMap(data -> {
        String json = SimpleJson.mapToJson(data);
        return response.writeStringAndFlush("data: " + json + "\n");
    });
}
```

Service layer asynchronous

- All "service" methods return an Observable<T>
- Enables the service layer implementation to:
  - block instead of using threads if resources are constrained
  - use multiple threads
  - use non-blocking IO
  - migrate an underlying implementation from network based to in-memory cache
  - no mutation of state is occurring that would cause thread-safety issues

RxJava

Anatomy



```
Observable.from(_doNetworkCall())
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(_resultObserver());
```

1 Observable

3 Schedule

2 Observer

Observable

Observer

4 Subscription

How It Work!

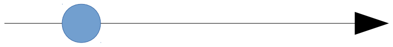
Observable<T>

compose & chain a stream

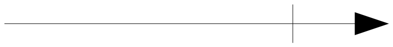
Consuming Observables

interface Observer<T>

onNext(T data)

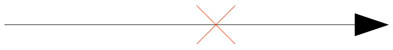


onCompleted()

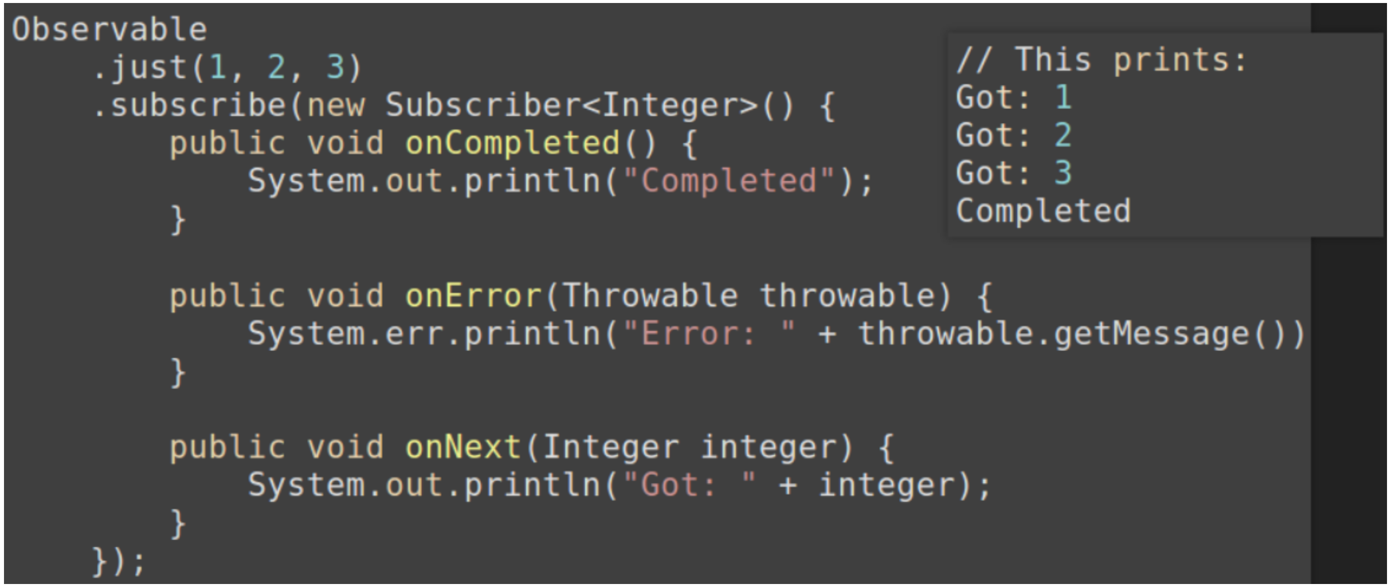


or

onError(Throwable t)



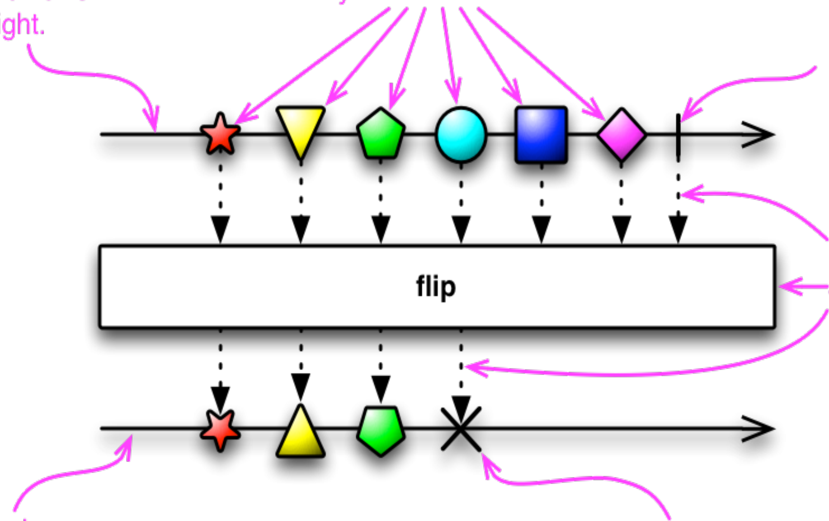
Consuming Observables



```
Observable
    .just(1, 2, 3)
    .subscribe(new Subscriber<Integer>() {
        public void onCompleted() {
            System.out.println("Completed");
        }

        public void onError(Throwable throwable) {
            System.err.println("Error: " + throwable.getMessage())
        }

        public void onNext(Integer integer) {
            System.out.println("Got: " + integer);
        }
    });
```

```
// This prints:
Got: 1
Got: 2
Got: 3
Completed
```

Marble Diagrams



This is the timeline of the Observable. Time flows from left to right.

These are items emitted by the Observable.

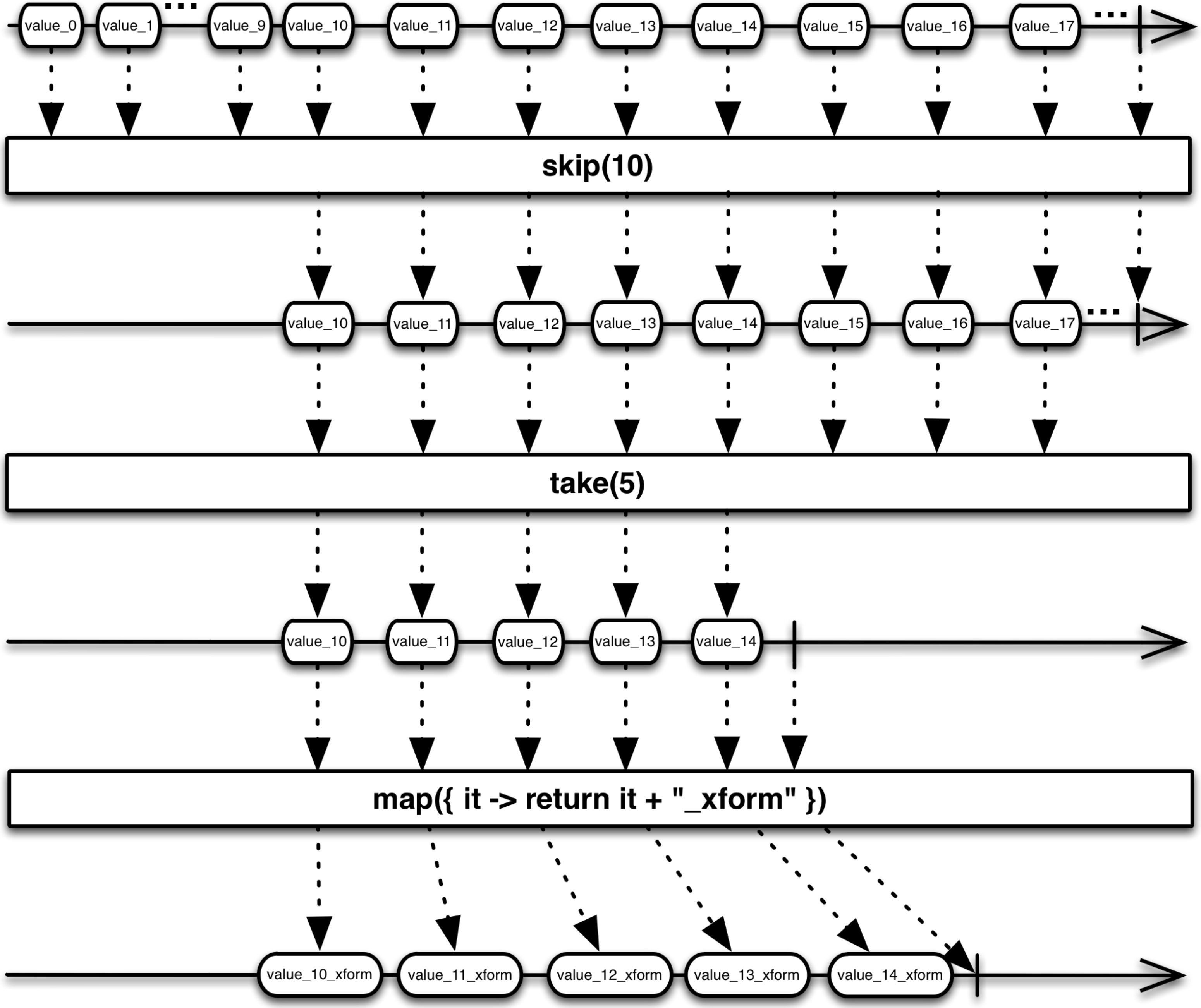
This vertical line indicates that the Observable has completed successfully.

These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.

This Observable is the result of the transformation.

If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

Compositions



1 Creation

Observable Factories

- just
- from
- empty / never / throw
- create

```
Observable.create(new OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        ...
    }
})
```

```
Observable.create(subscriber -> {
     subscriber.onNext("Hello world");
     subscriber.onCompleted();
})
```

```
Observable.create(subscriber -> {
     subscriber.onNext("Hello");
     subscriber.onNext("world");
     subscriber.onNext("!");
     subscriber.onCompleted();
})
```

```
Observable.create(subscriber -> {
    int i = 0;
    while (!subscriber.isUnsubscribed()) {
        subscriber.onNext(i++);
    }
})
```

Create from existing method

```java
public String value() {
    return ...;
}
```

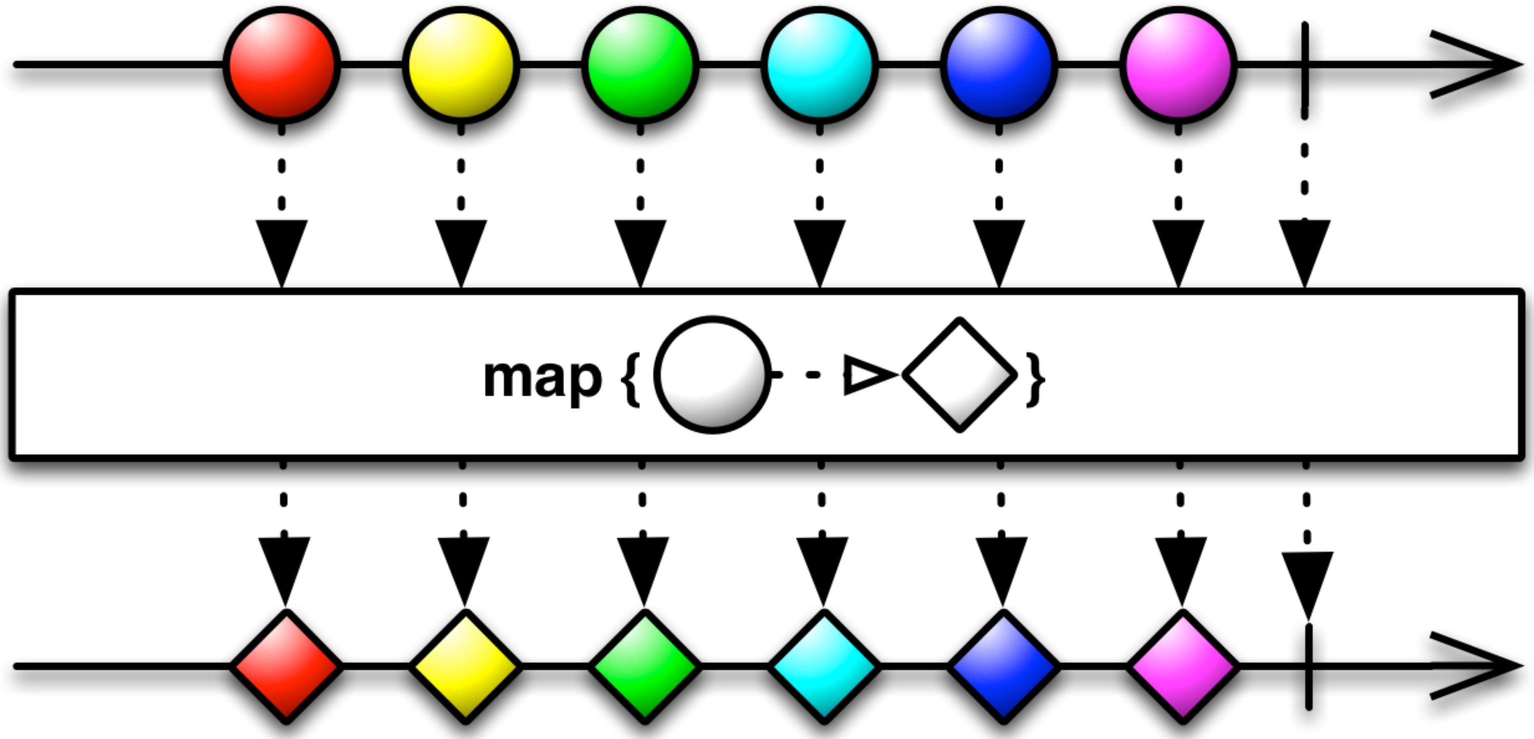
```java
public Observable<String> valueObservable() {
   return Observable.defer(() -> Observable.just(value()));
}
```

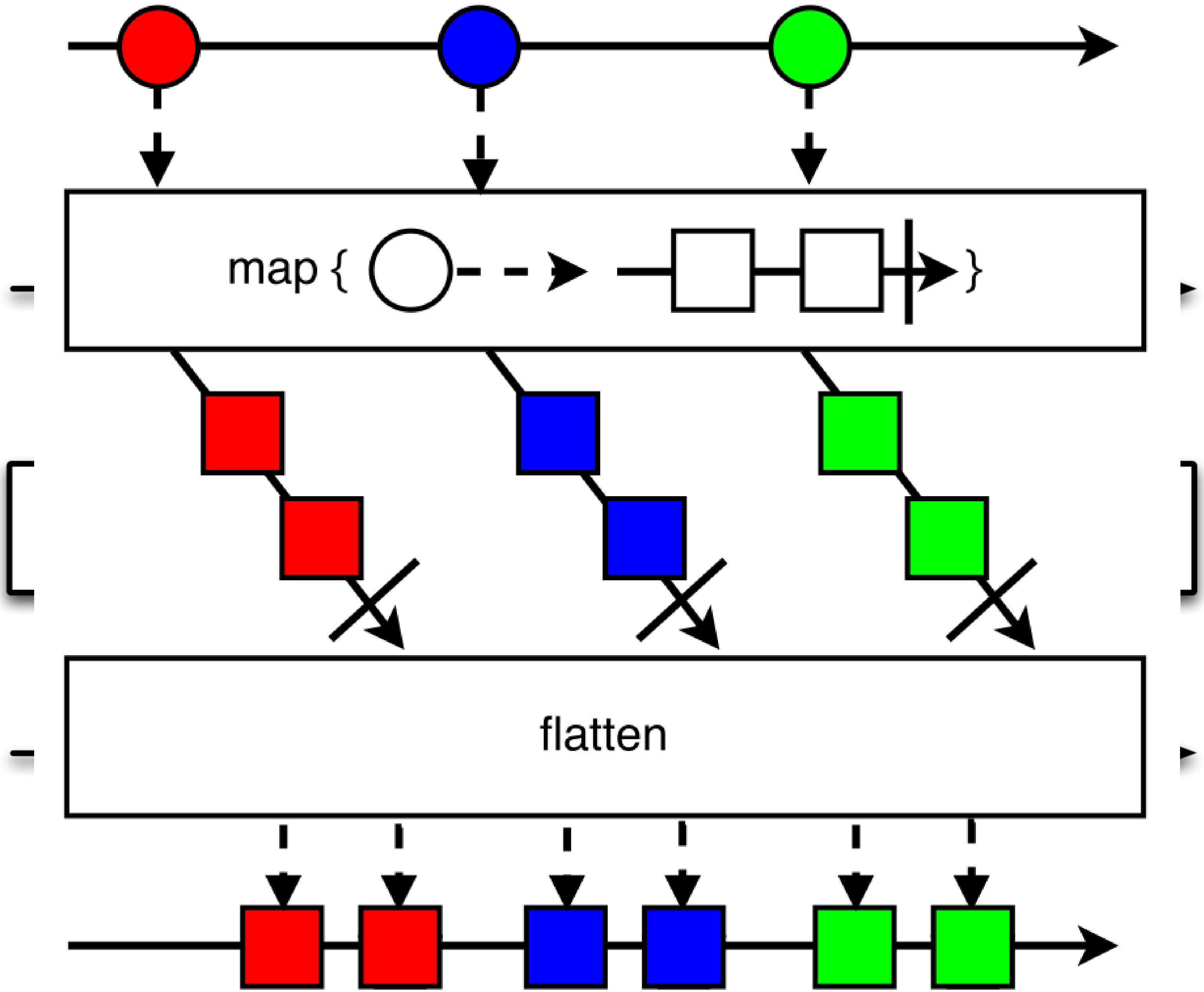
```java
public Observable<String> valueObservable() {
   return Observable.just(value());
}
```

```java
public Observable<String> valueObservable() {
   return Observable.fromCallable(this::value);
}
```

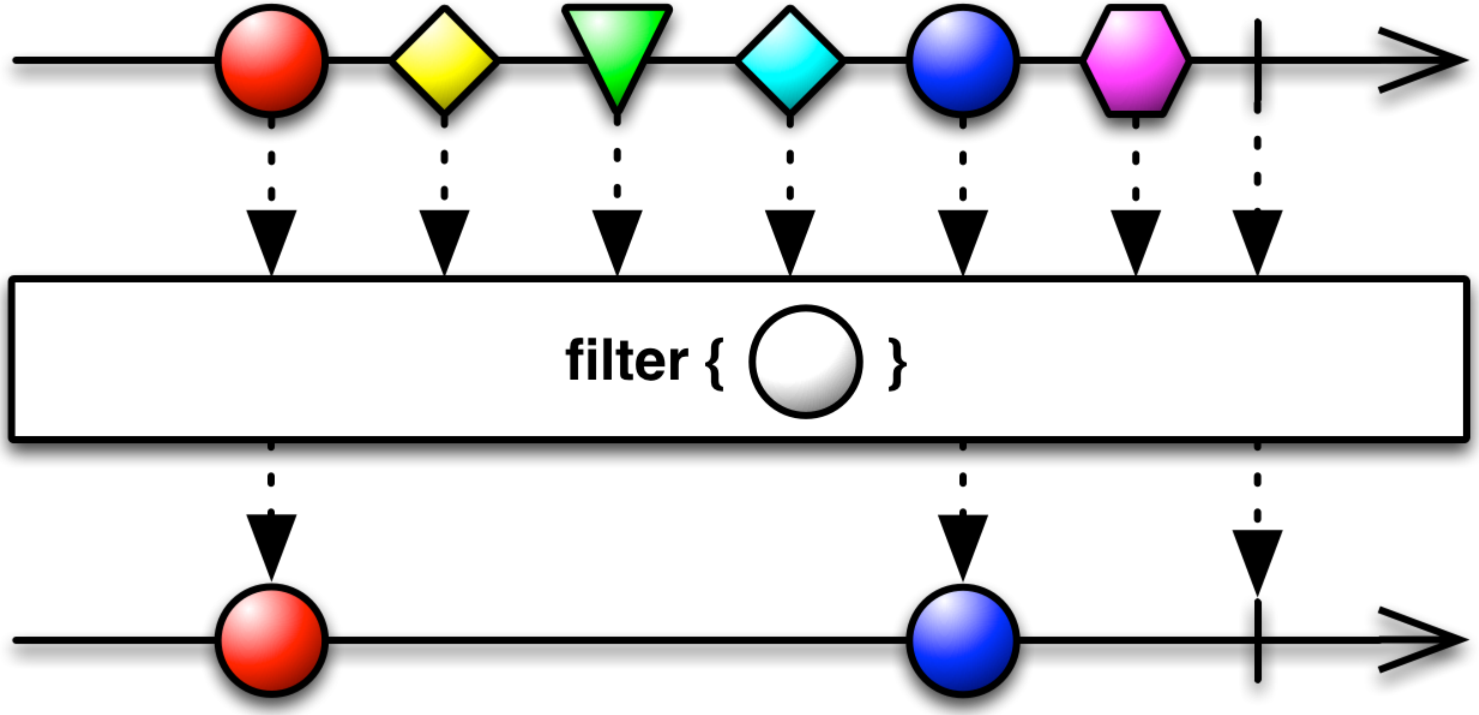
```java
public Observable<String> valueObservable() {
    return Observable.create(subscriber -> {
        try {
            subscriber.onNext(value());
            subscriber.onCompleted();
        } catch (Exception e) {
            subscriber.onError(e);
        }
    });
}
```

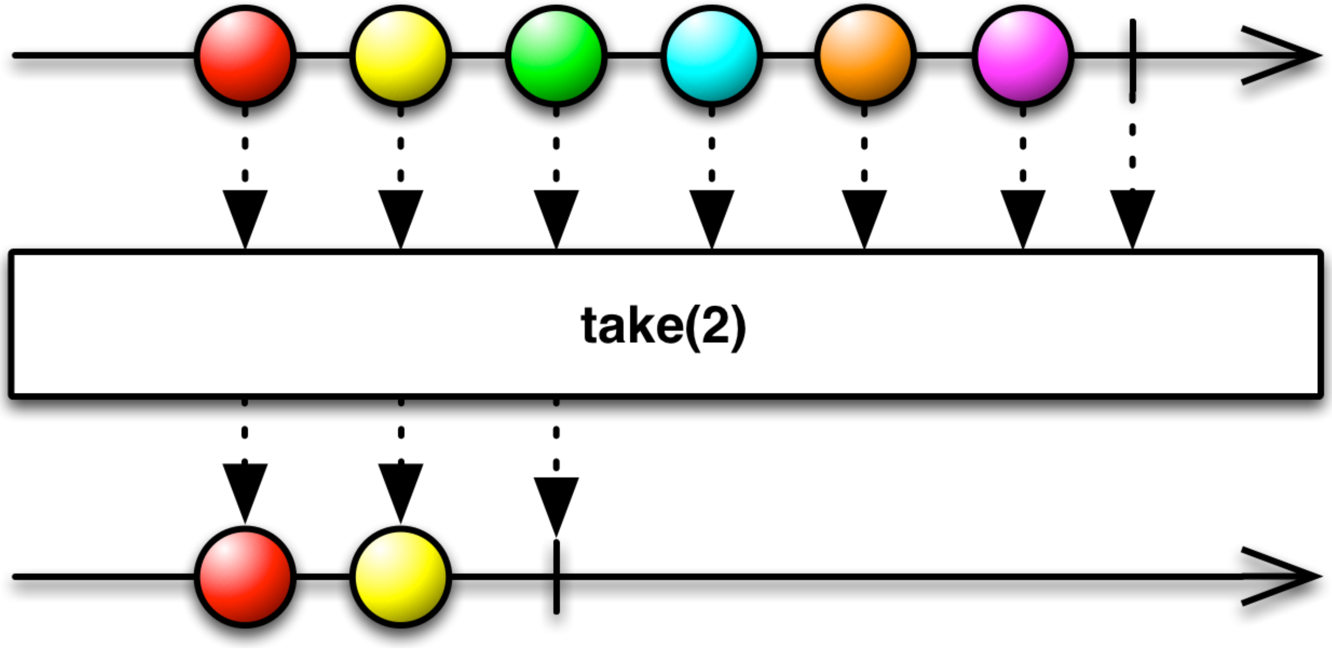
2 Transform


map { }

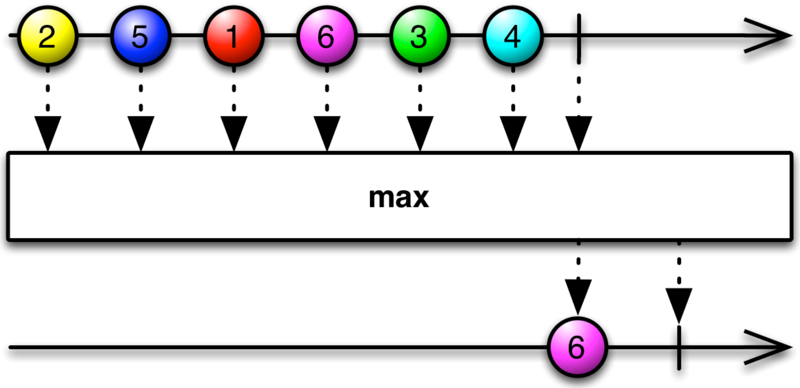

map { }
flatten

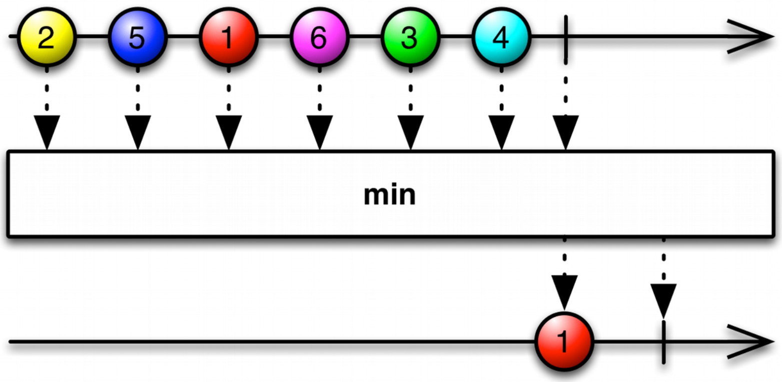
3 Filter

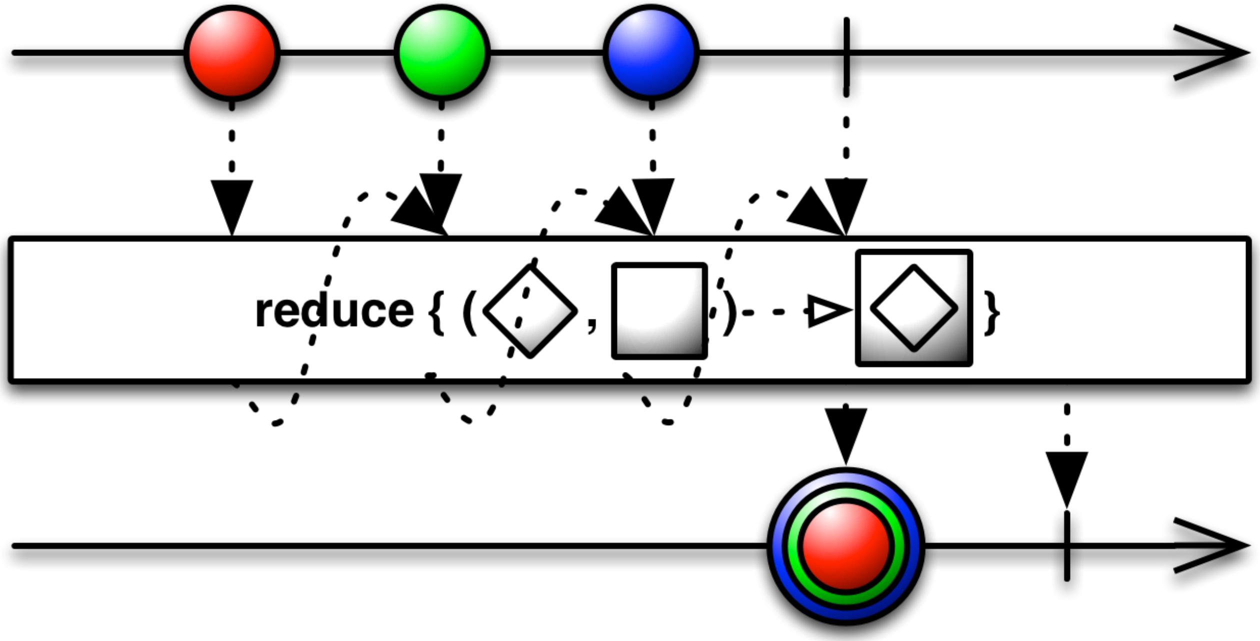

filter { ○ }


take(2)

4 Aggregate


2
5
1
6
3
4
max
6


2
5
1
6
3
4
min
1

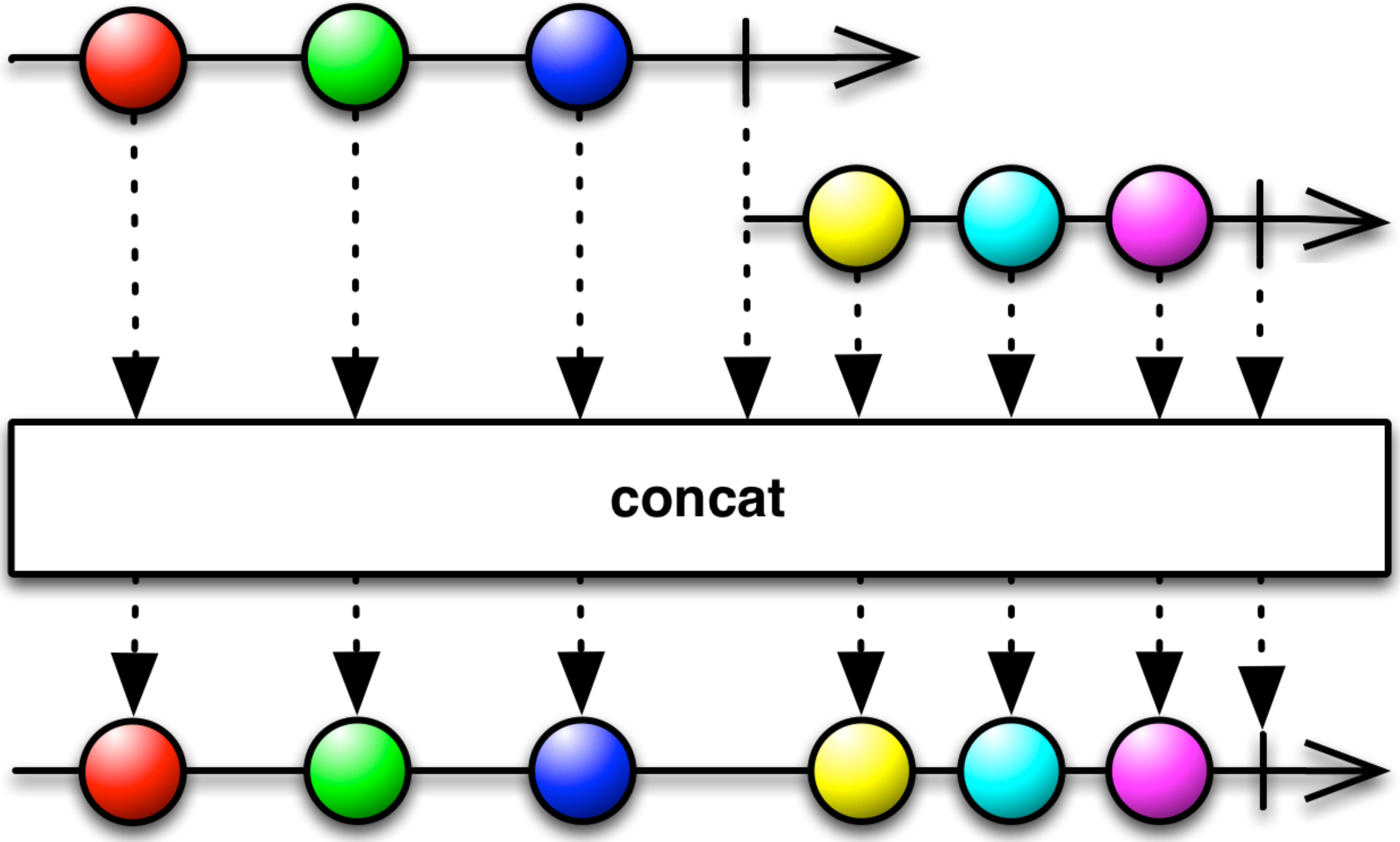

reduce { (◇, □) - - ▷ ◇ }

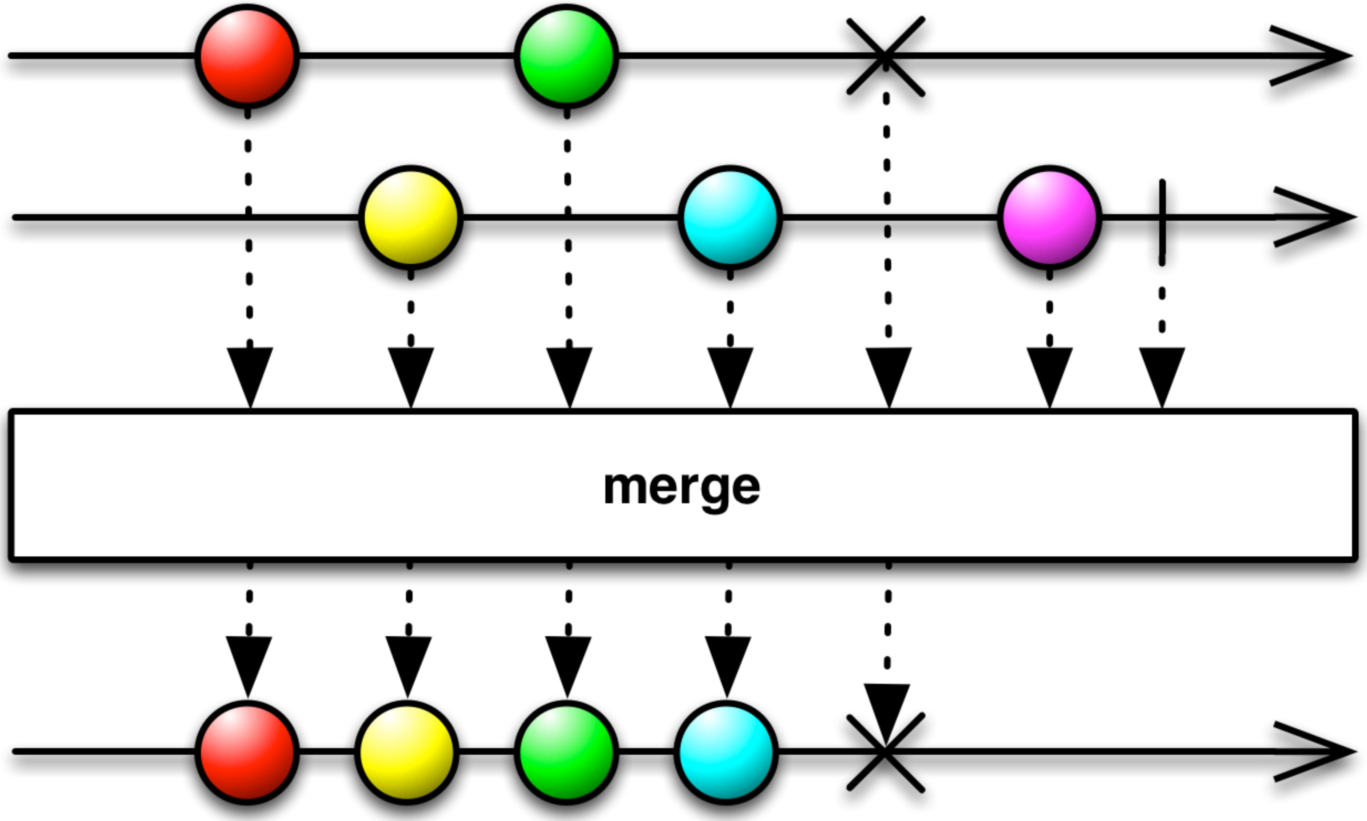
5 Side effects

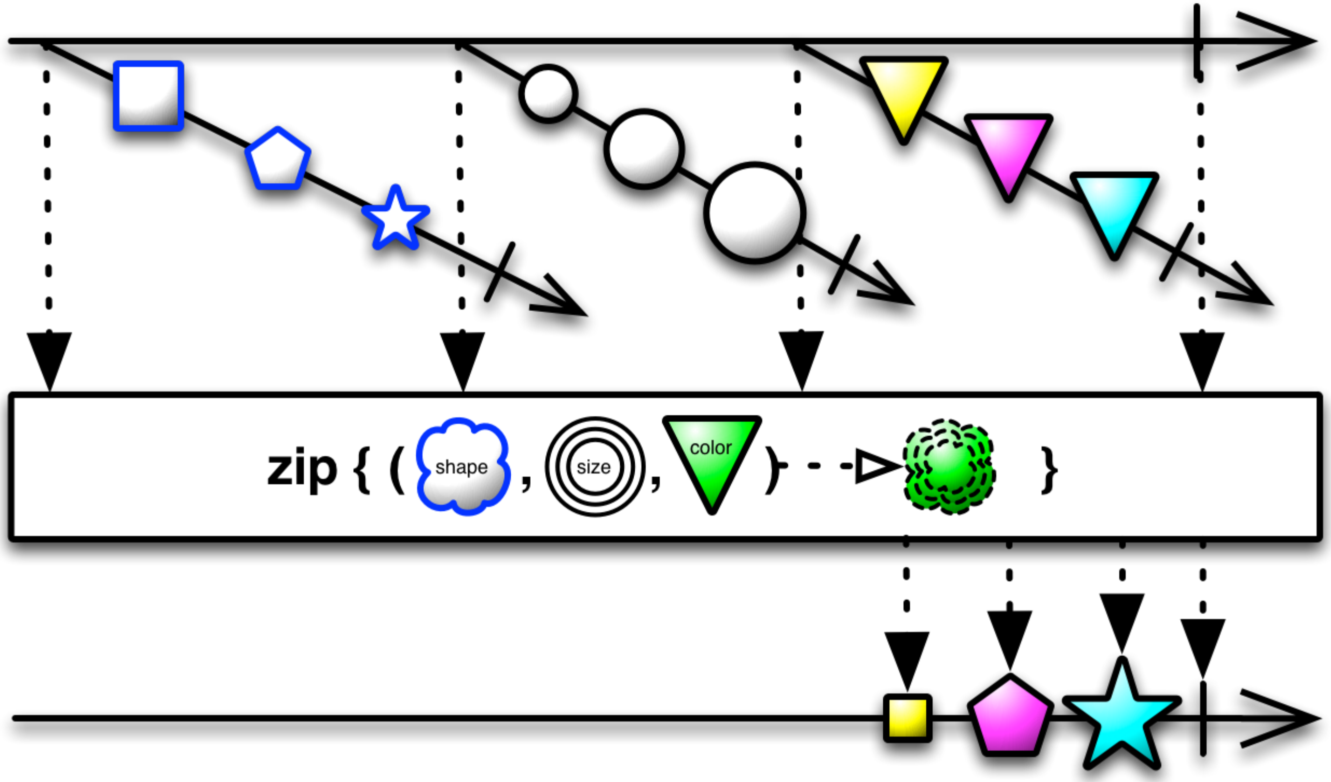
doOnXX

- doOnNext
- doOnError
- doOnCompleted
- doOnEach

6 Combine


concat

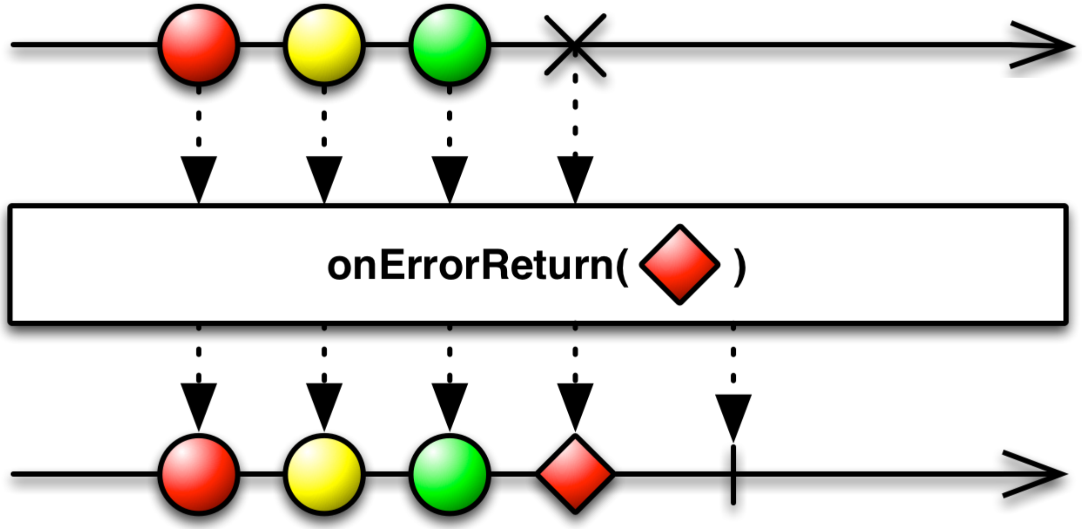

merge

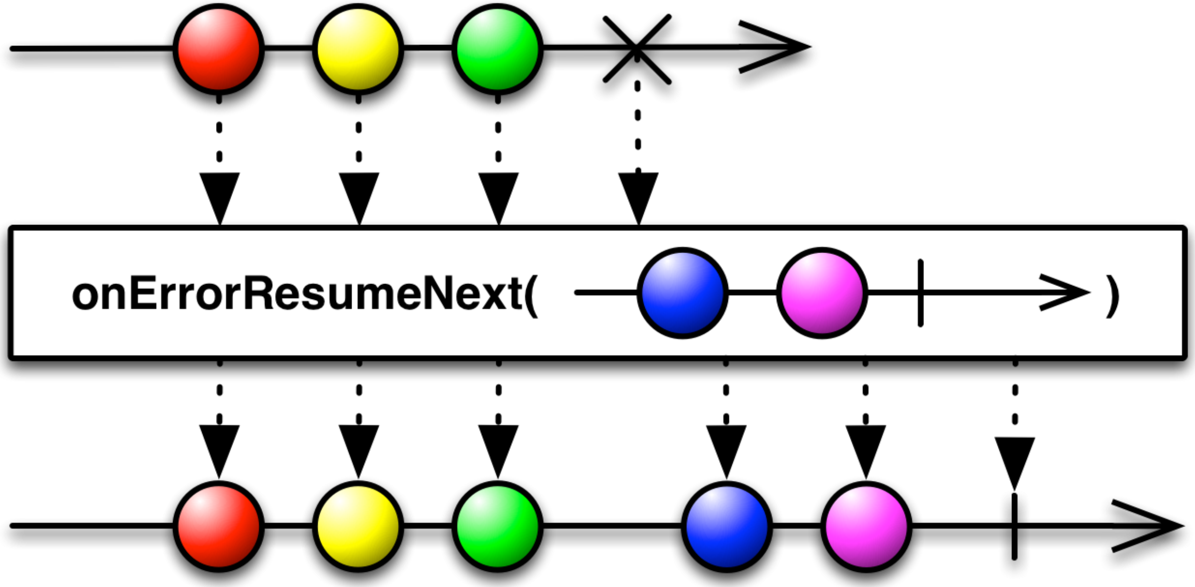

zip { (
shape
,
size
,
color
) - - ▷
}

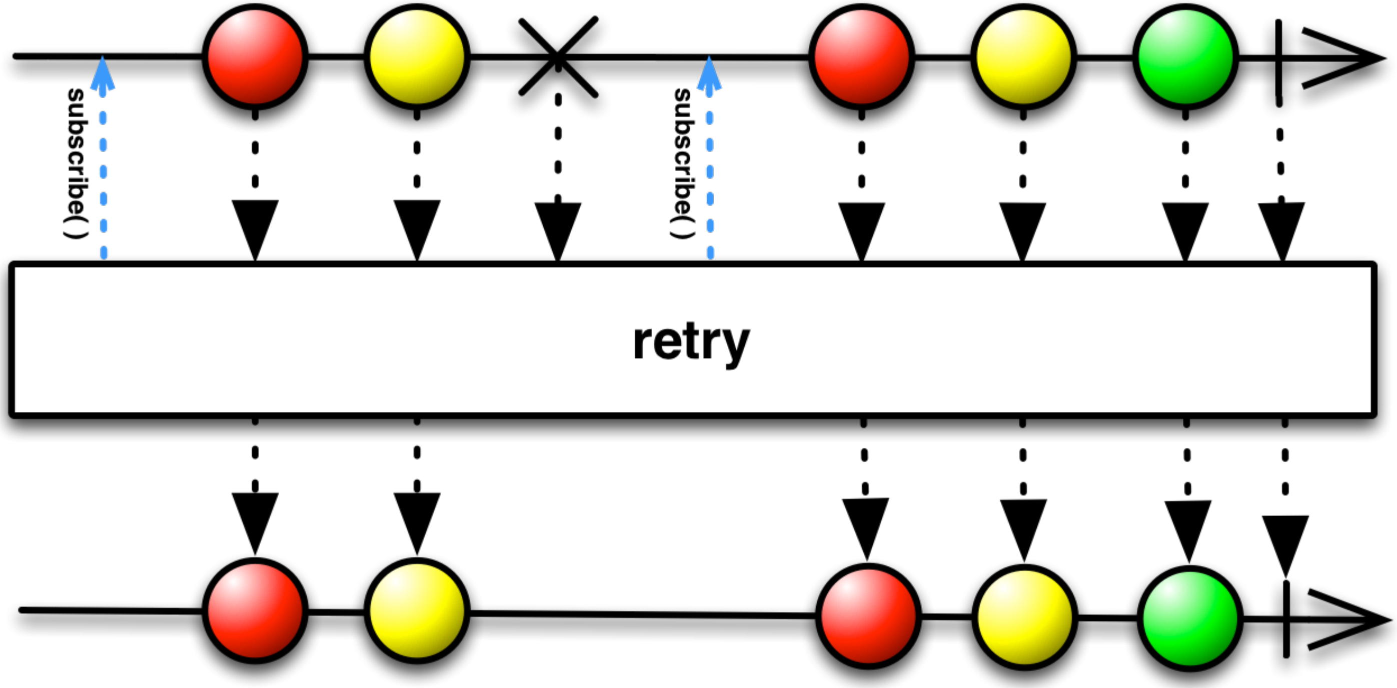
7 Recover / Retry

Error Handling

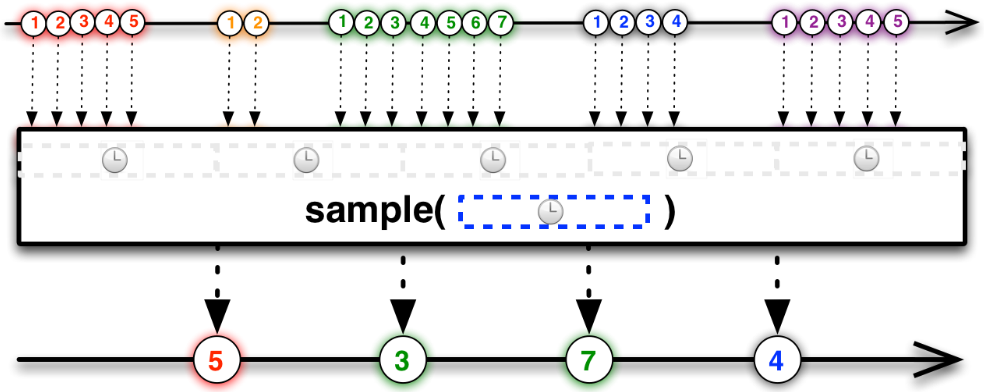
- Return a default value instead
- Flip over to a backup Observable
- Retry the Observable (immediately or with backoff)


onErrorReturn( )

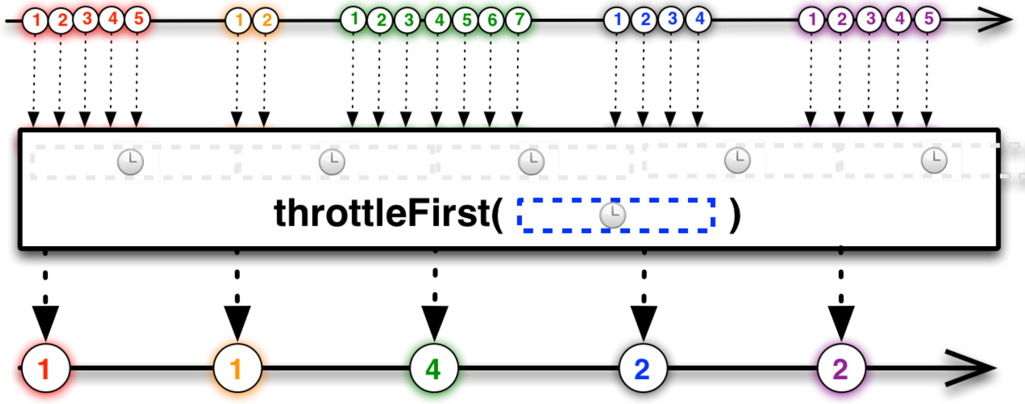

onErrorResumeNext( )

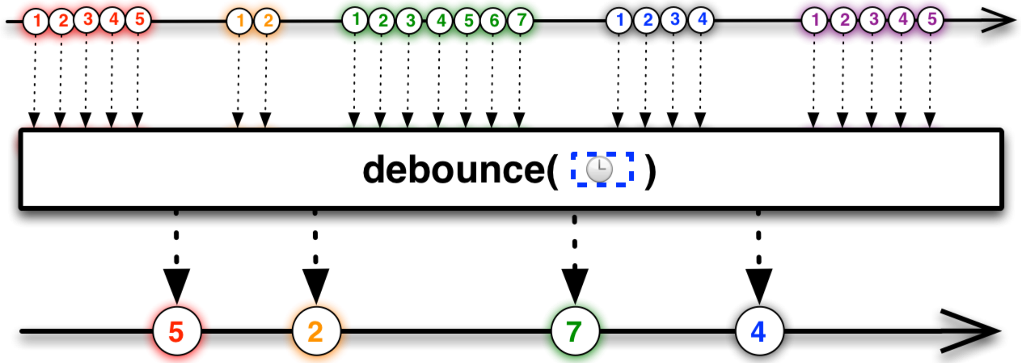

subscribe( )
subscribe( )
retry

8 Backpressure

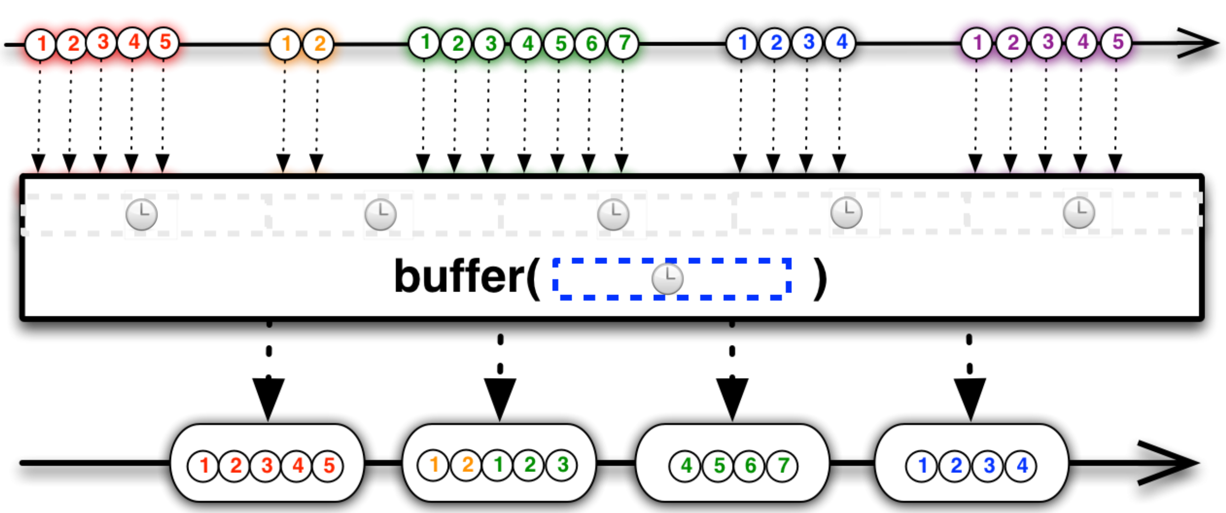


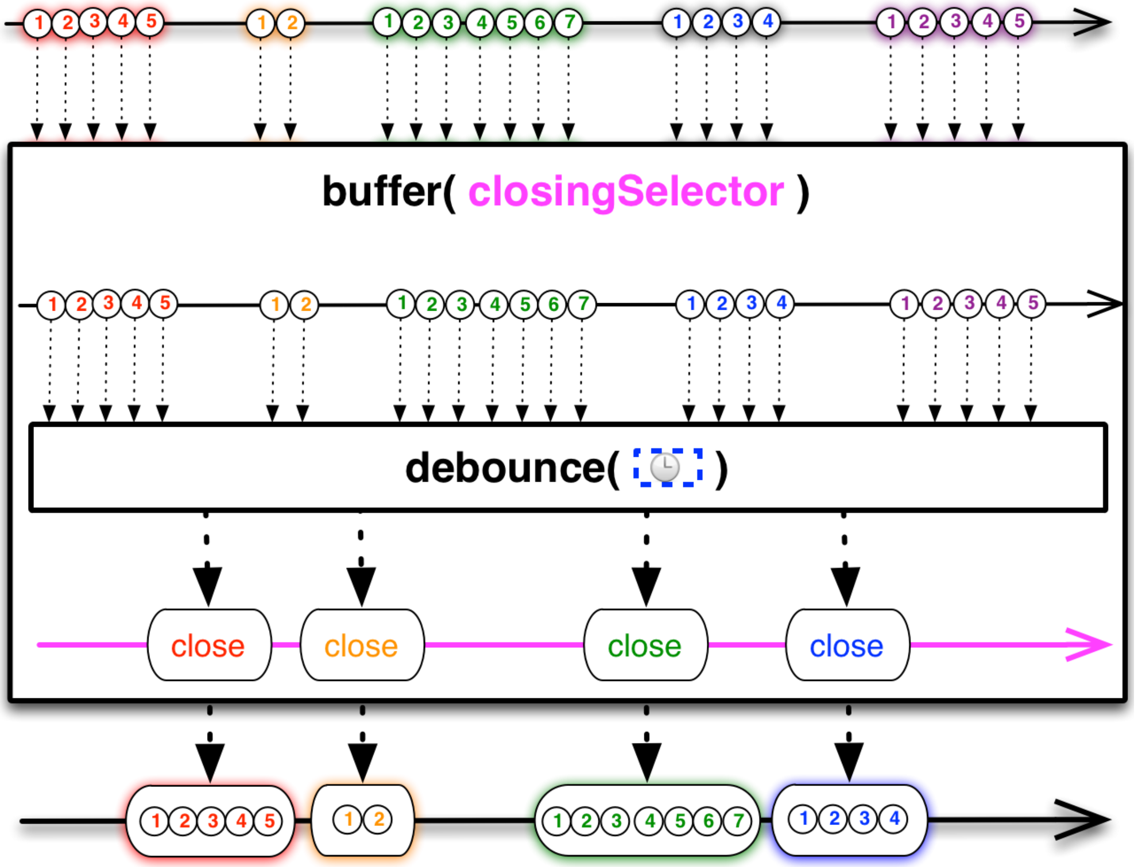
(or throttleLast)



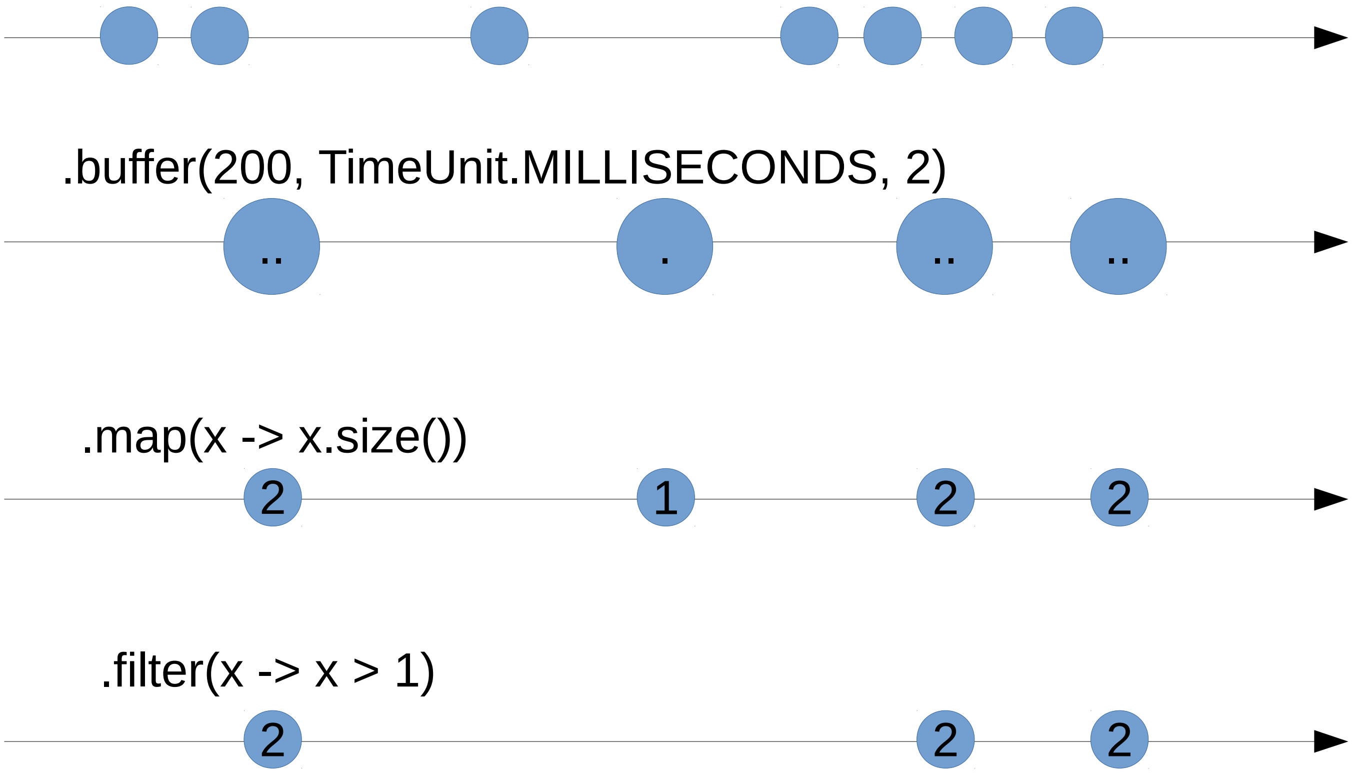


(or throttleWithTimeout)


buffer( )


buffer(closingSelector)
debounce( )
close
close
close
close

Mouse Clicks



.buffer(200, TimeUnit.MILLISECONDS, 2)

.map(x -> x.size())

.filter(x -> x > 1)

More Operation

Scheduler

- RxJava is synchronous by default
- but work can be defined asynchronously using schedulers.
- Schedulers io(), computation(), newThread(), immediate()
- AndroidSchedulers mainThread()

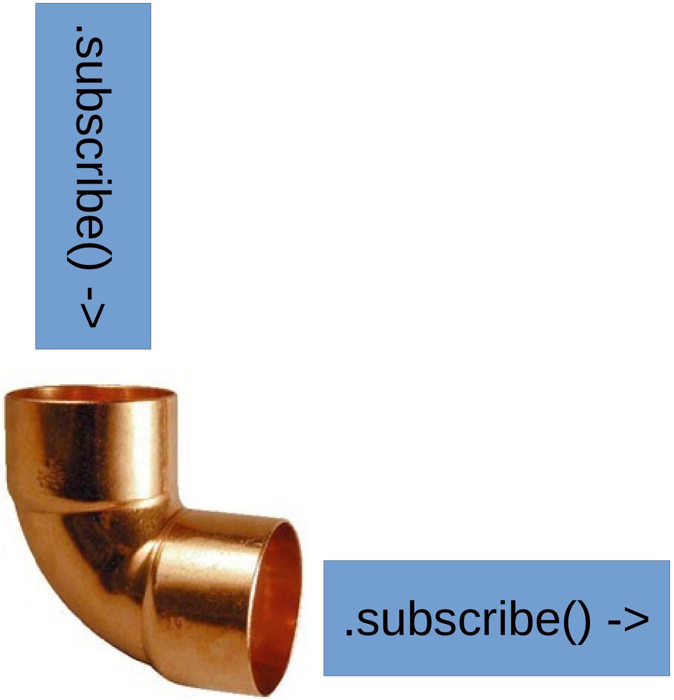
```
Observable.from(_doNetworkCall())
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(_resultObserver());
```

Multicasting in RxJava

- Use a ConnectableObservable (via publish() or replay())
- Use a Subject

Subject

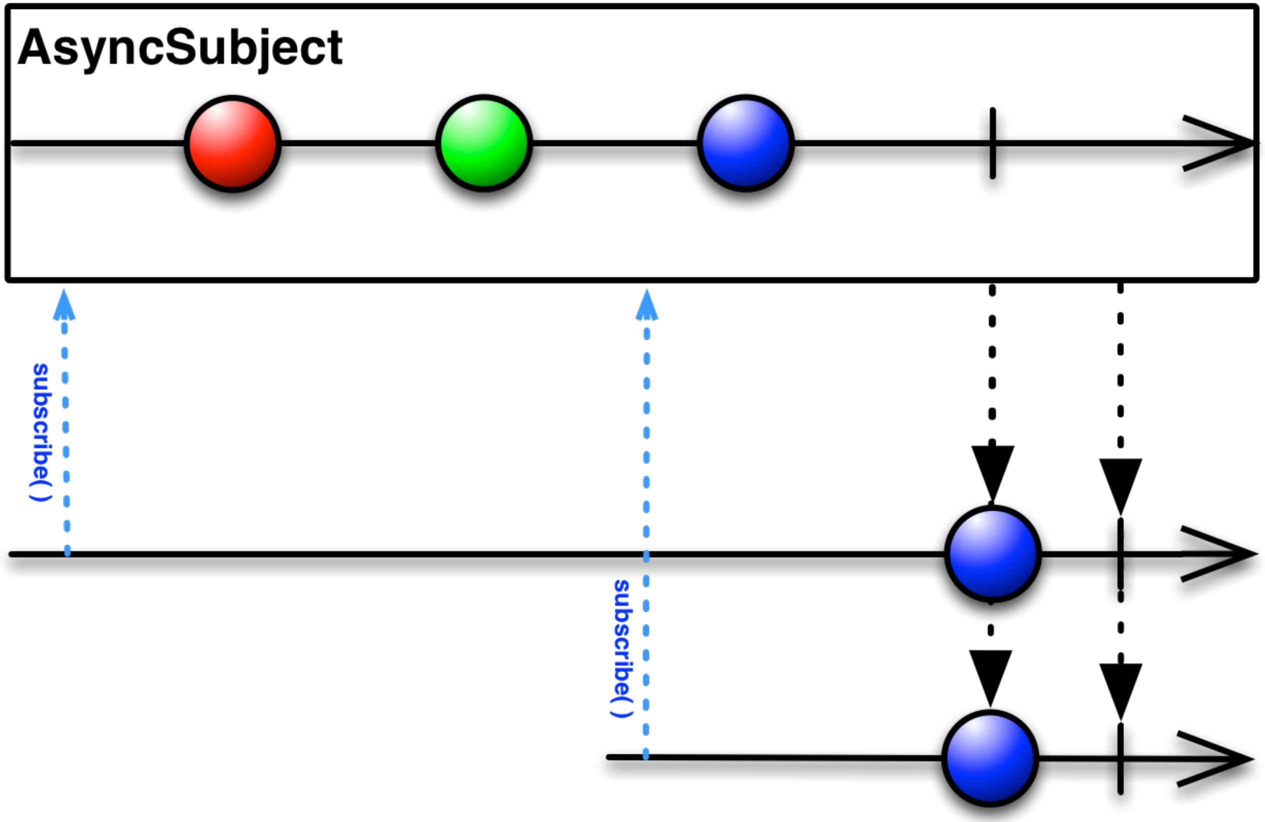
Subject



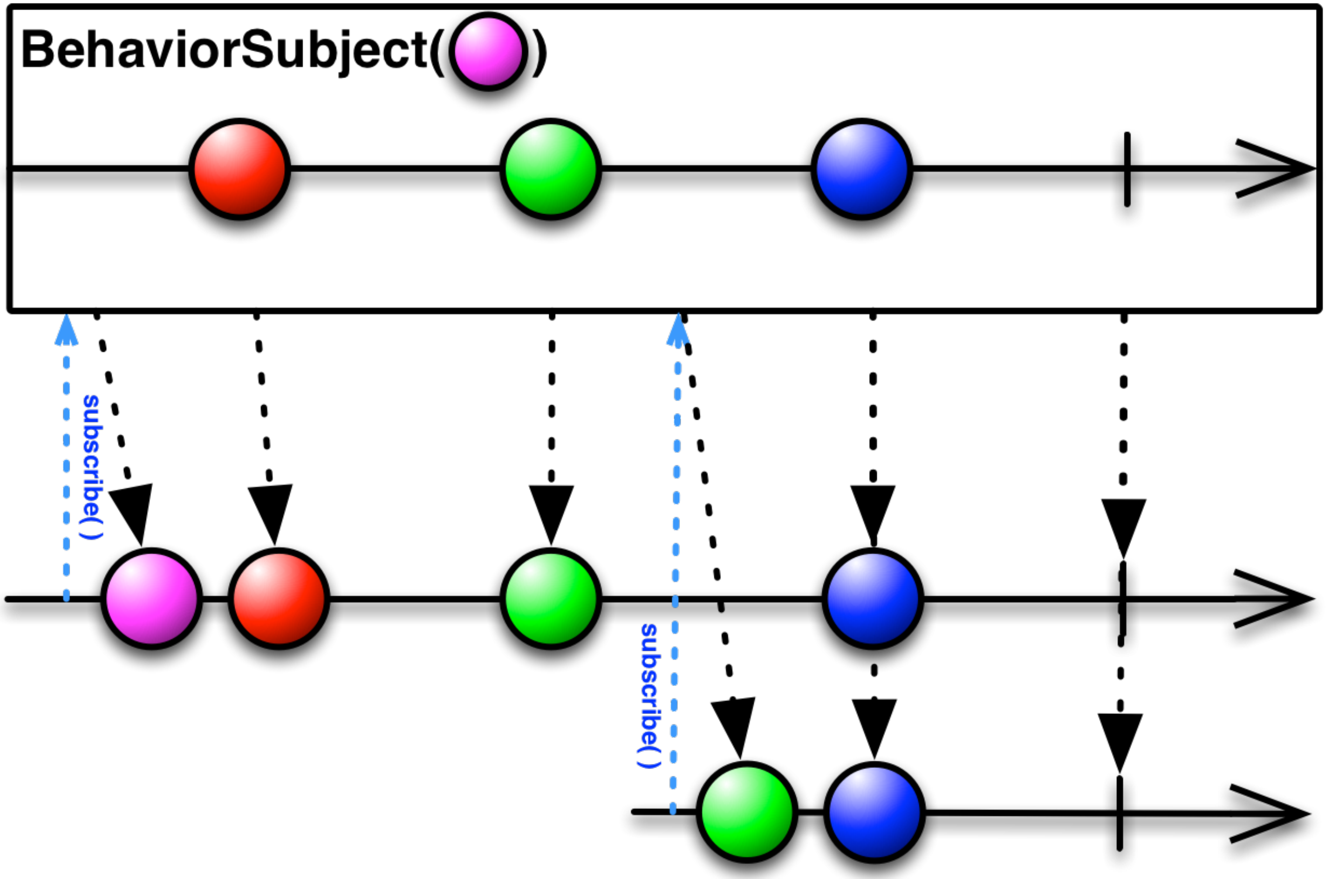
.subscribe() ->

.subscribe() ->

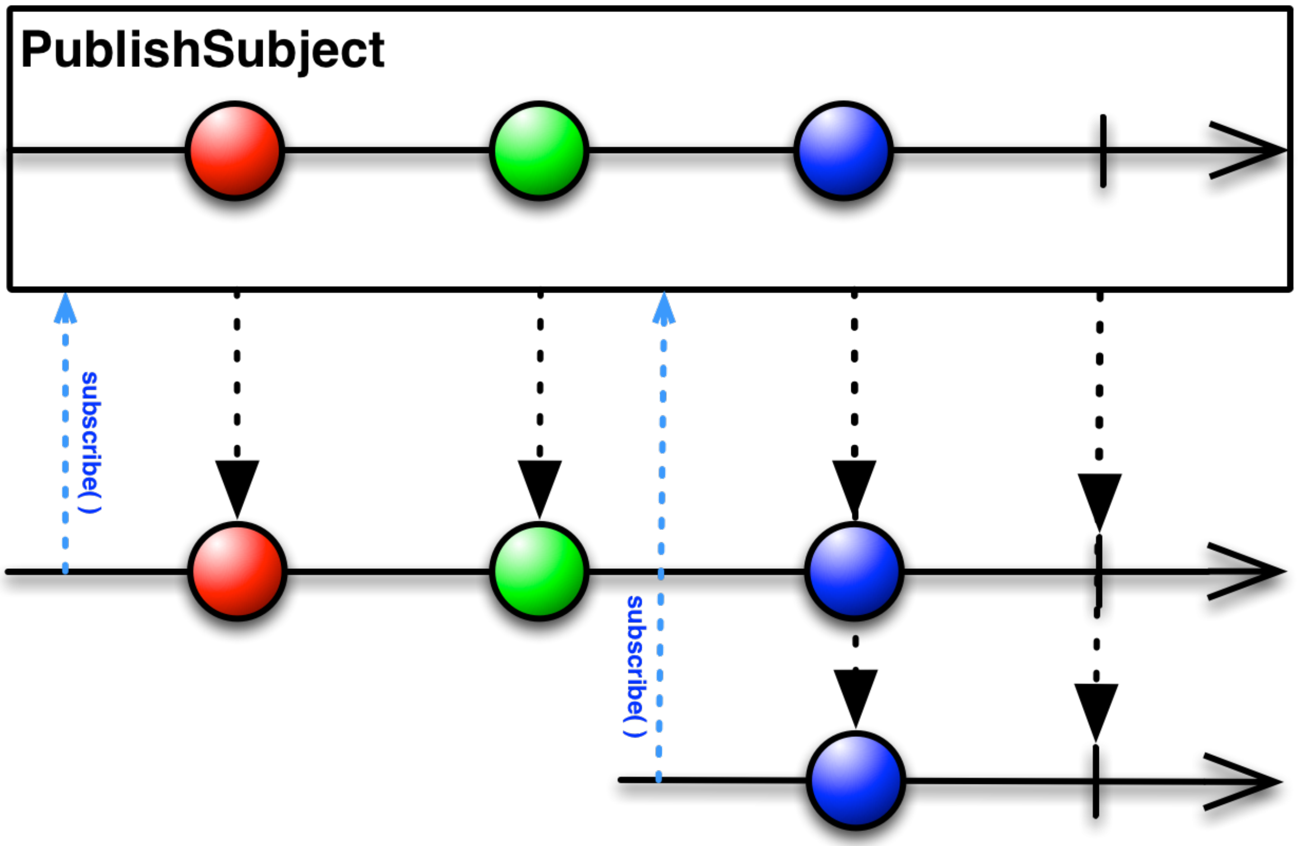
AsyncSubject



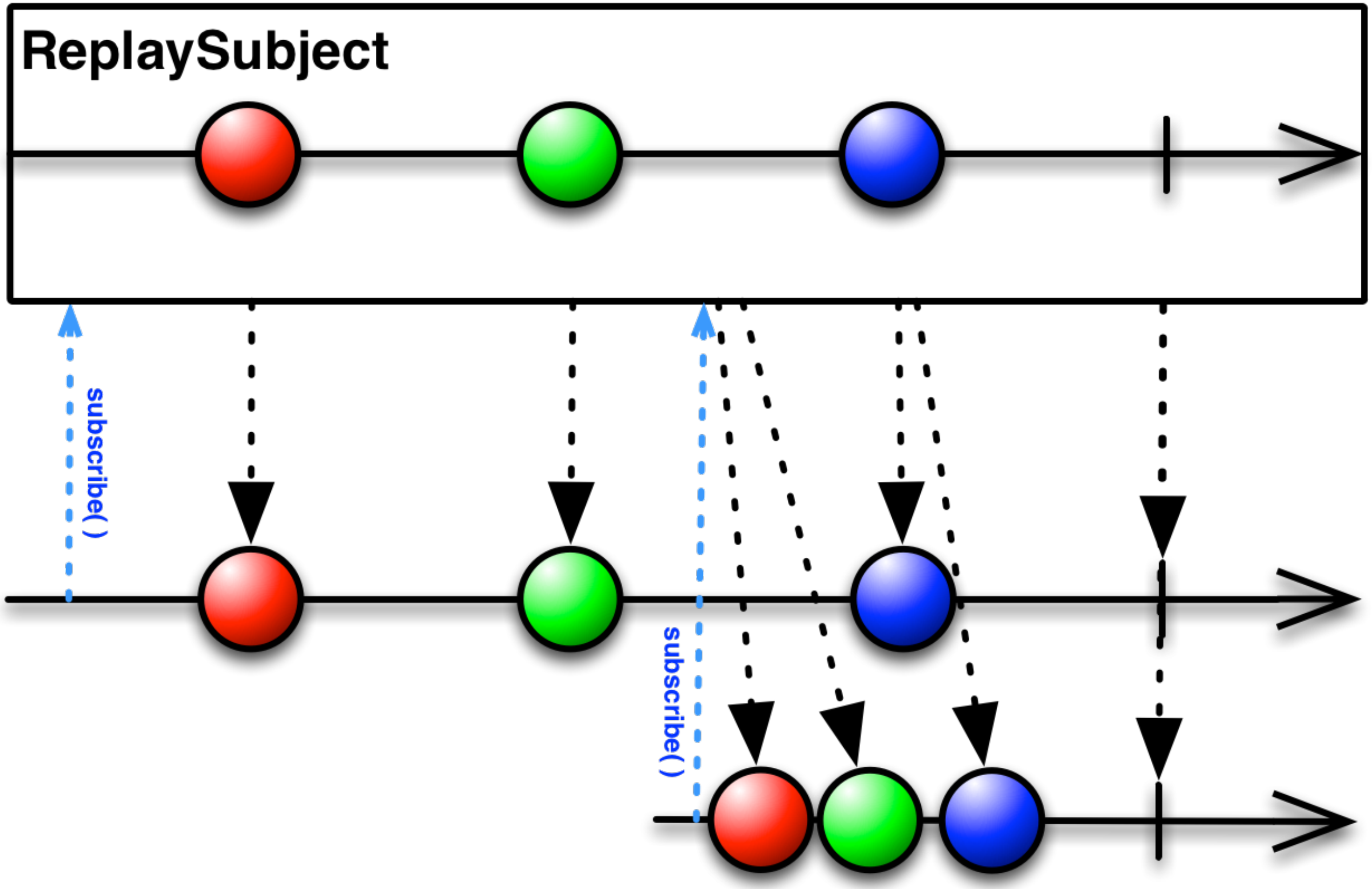
BehaviorSubject



PublishSubject



ReplaySubject



Thank You