# Reactive Programing with RxJava

Piya Lumyong

Why?

Blocking
it’s Evil

we need

parallelizable

asynchronous code

but

How ?

## Callbacks?

no composition

Callback Hell

## Futures<T>?

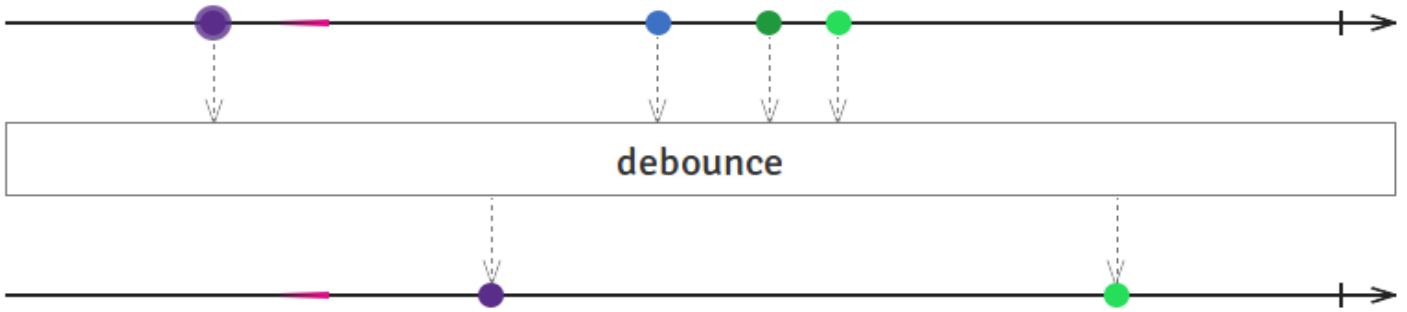
to easy to block (get)

complex beyond 1 level
of composition


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() |

# We use ReactiveX













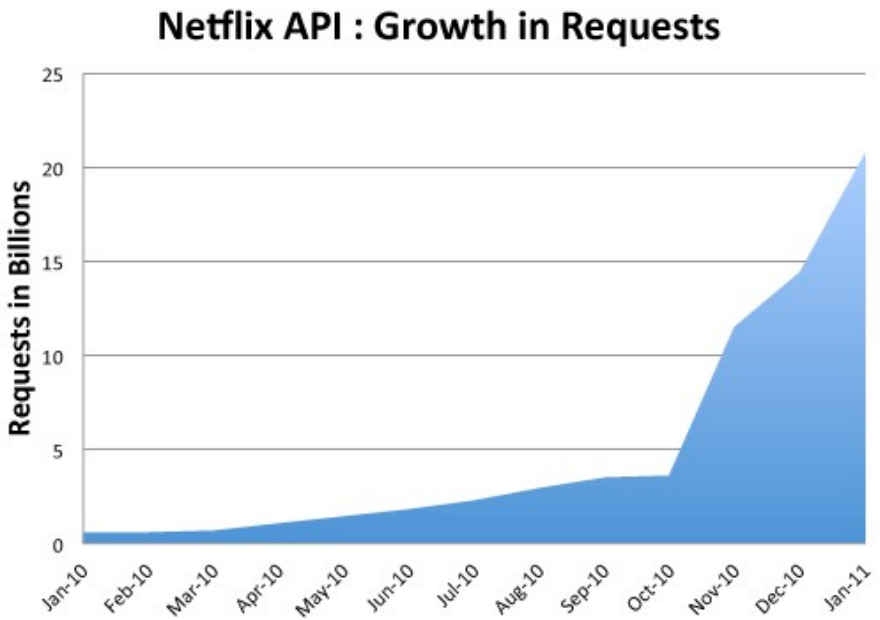
# RxJava :

is a Java VM implementation of Reactive Extensions (ReactiveX)

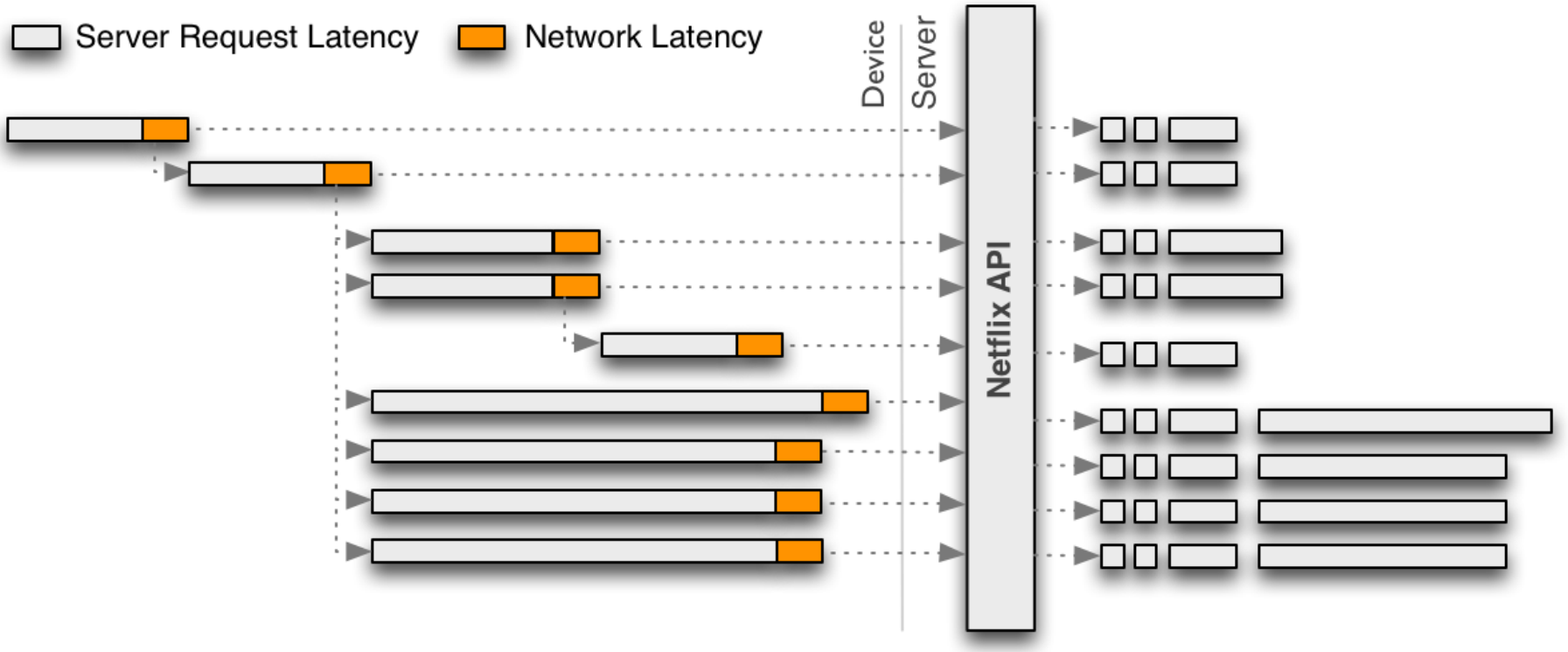
# RxJava By NETFLIX

- 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

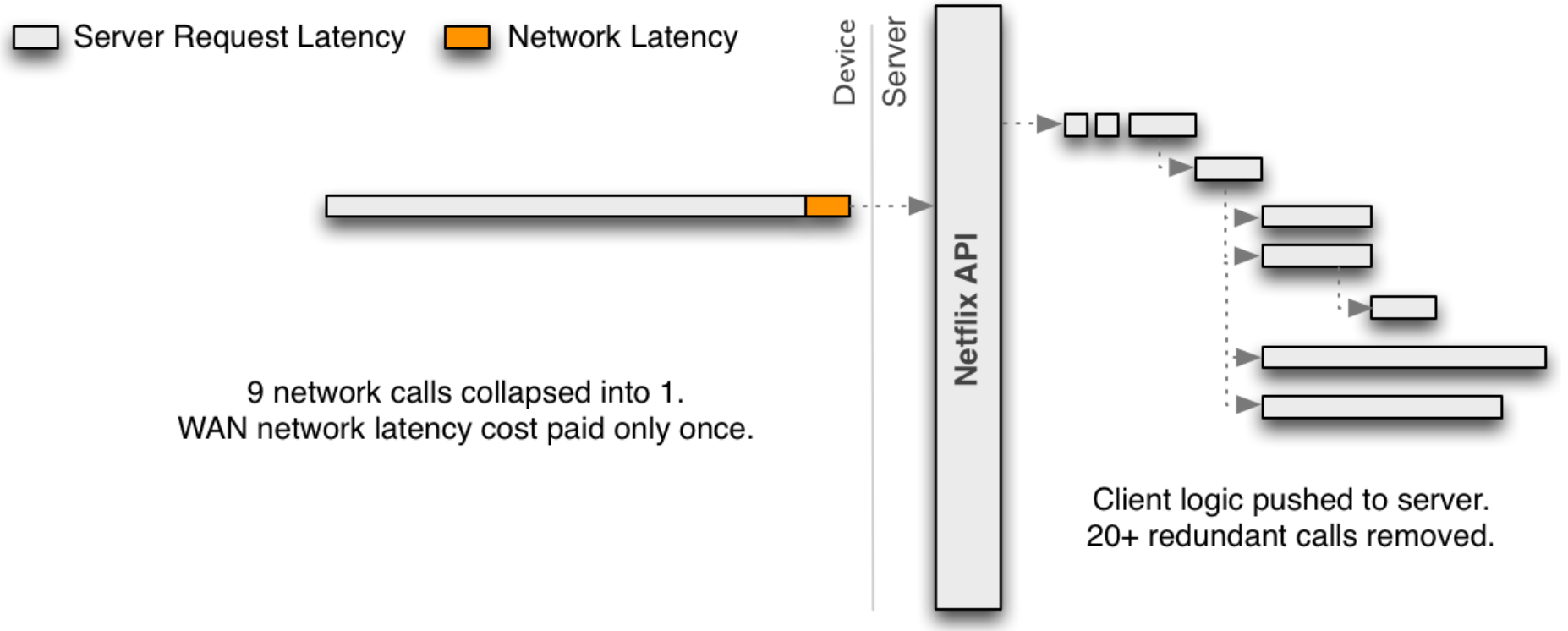
# Redesigning the Netflix API



# Client/Server communication



# Reduce Chattiness



# 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($ -> {
            return s.getDataAsMap();
        });

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

# RxJava

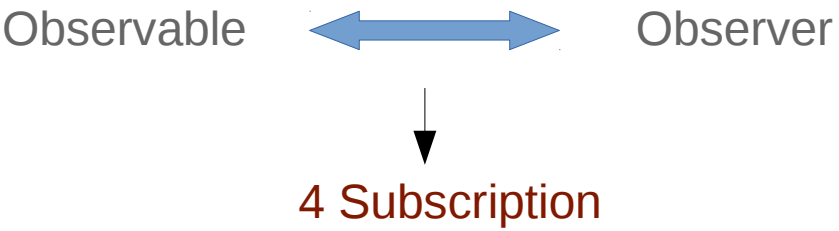
# Anatomy

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

1 Observable

3 Schedule

2 Observer



# How It Work!

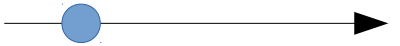
# Observable<T>

compose & chain a stream

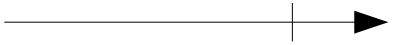
Consuming Observables

# interface Observer<T>

onNext(T data)



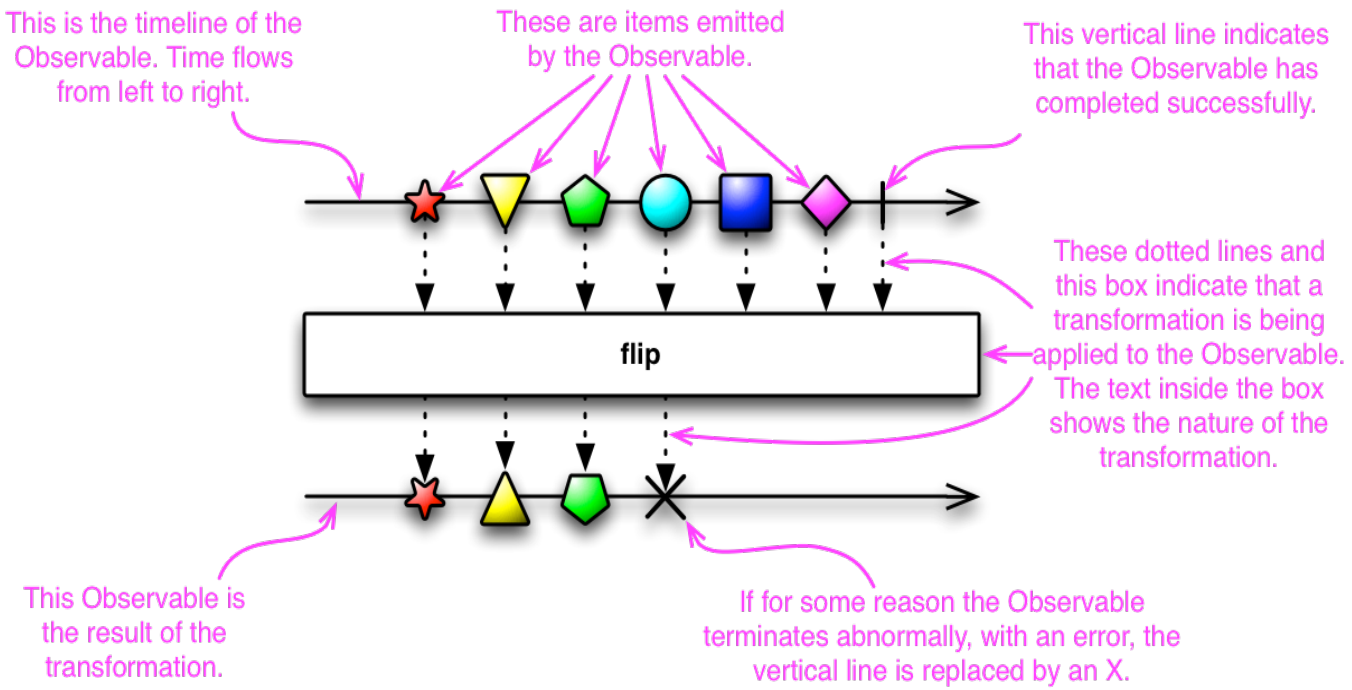
onCompleted()



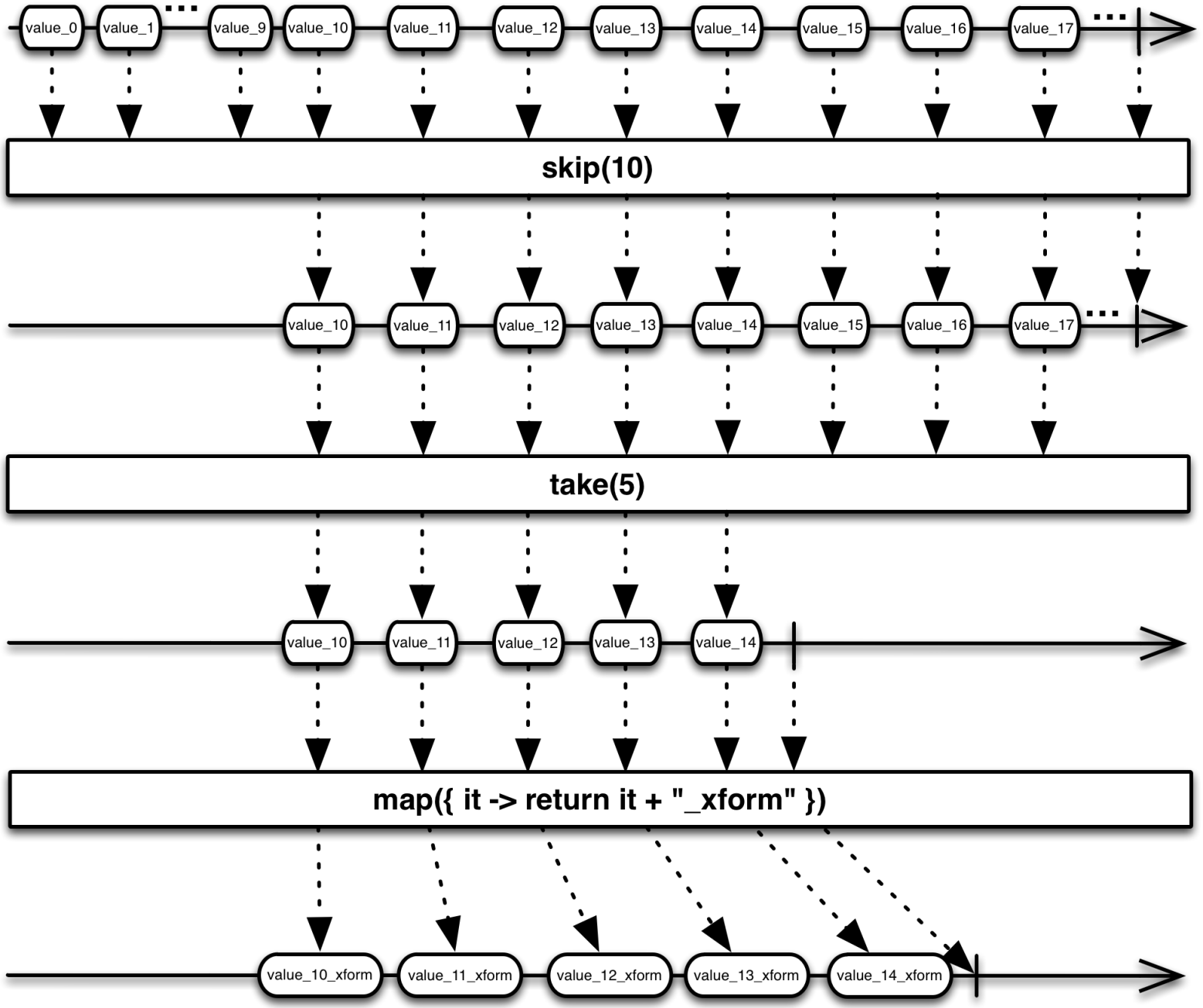
or

onError(Throwable t)

# Marble Diagrams



# 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

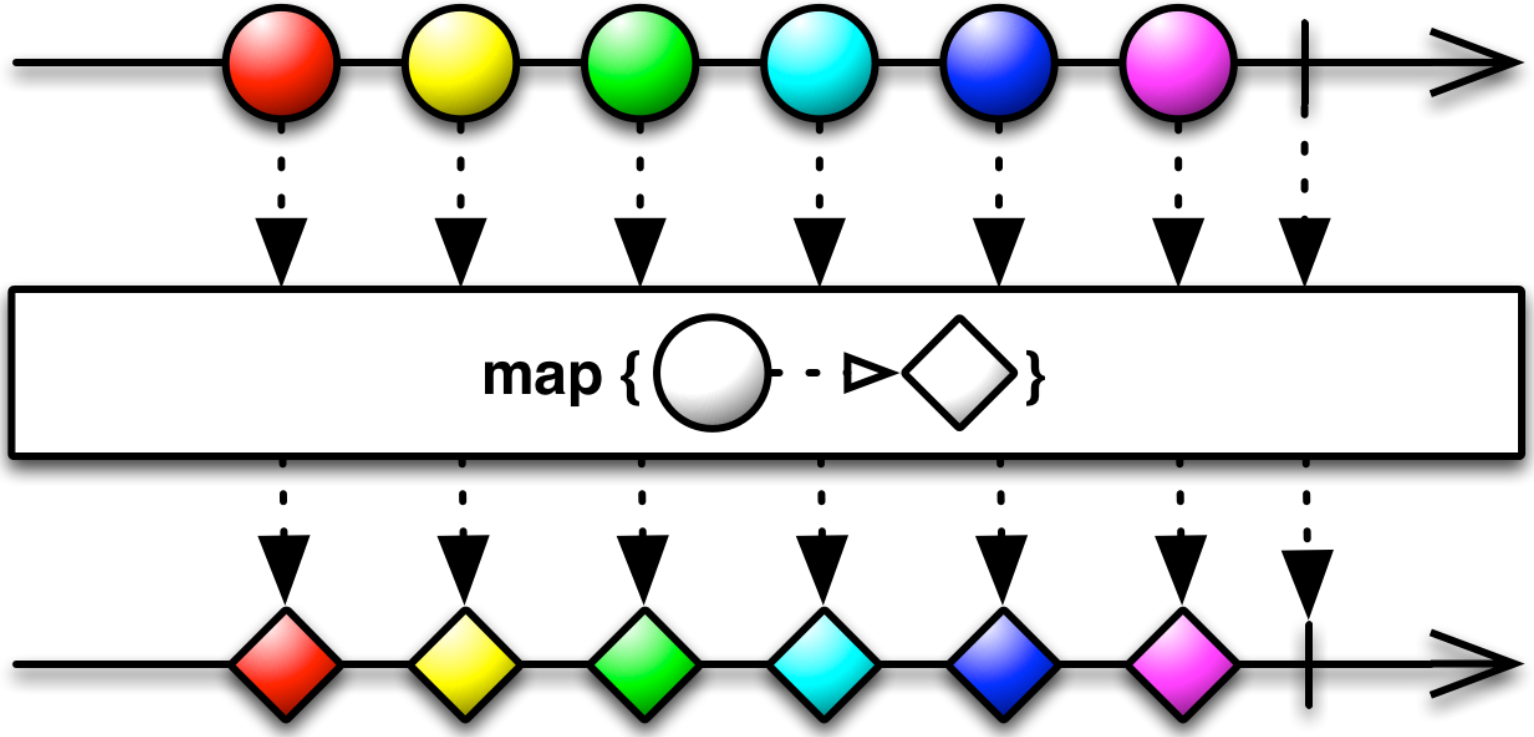
```
public String value() {
    return ...;
}
```

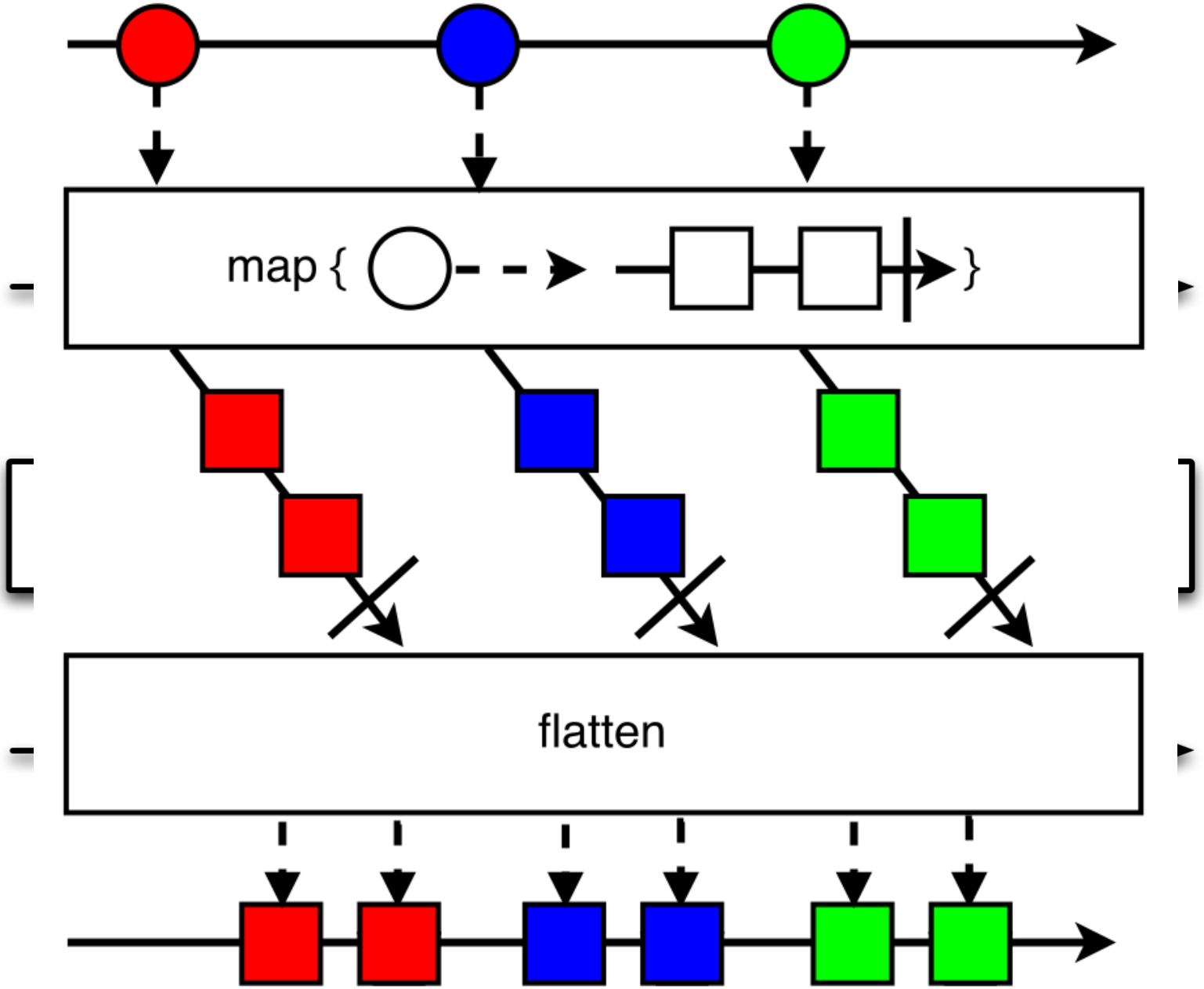
```
public Observable<String> valueObservable() {
    return Observable.just(value());
}
```

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

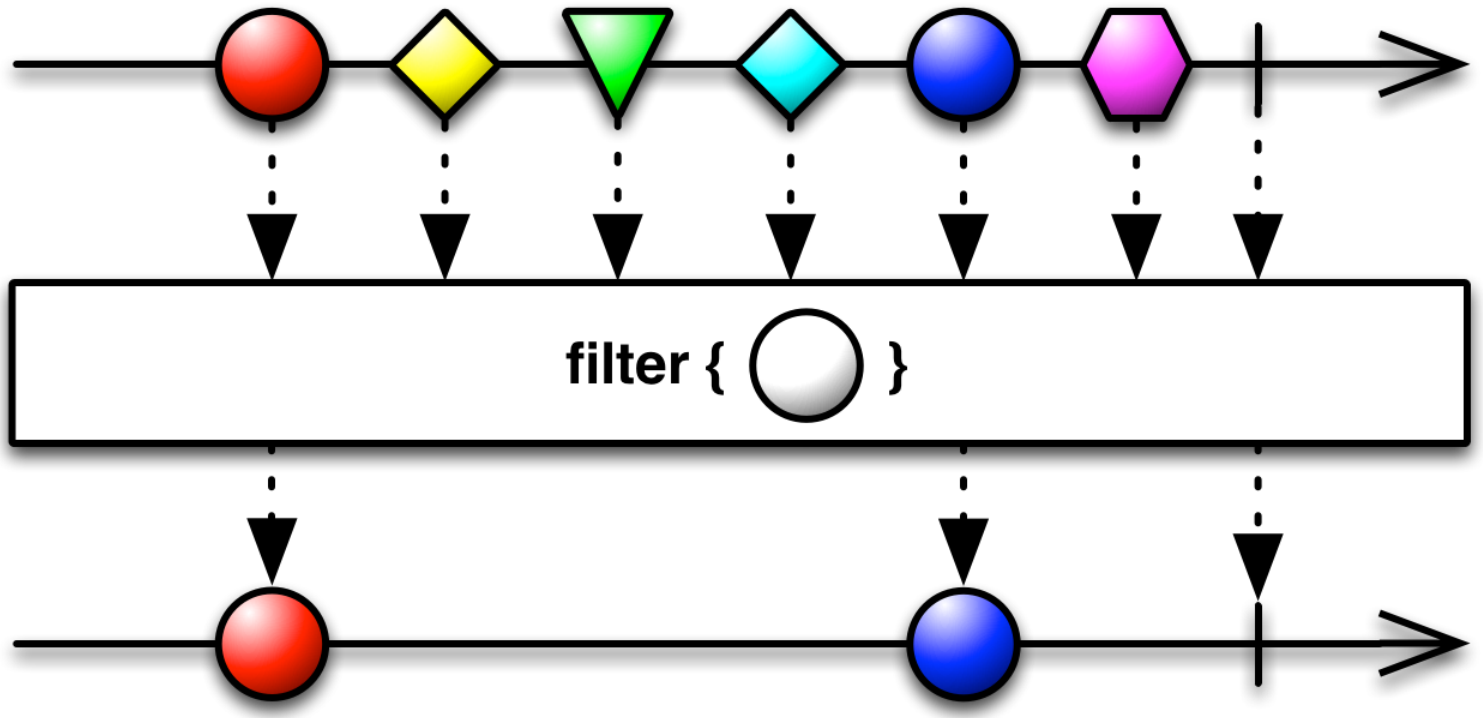
```
public Observable<String> valueObservable() {
    return Observable.create(subscriber -> {
        try {
            subscriber.onNext(value());
            subscriber.onCompleted();
        } catch (Exception e) {
            subscriber.onError(e);
        }
    });
}
```

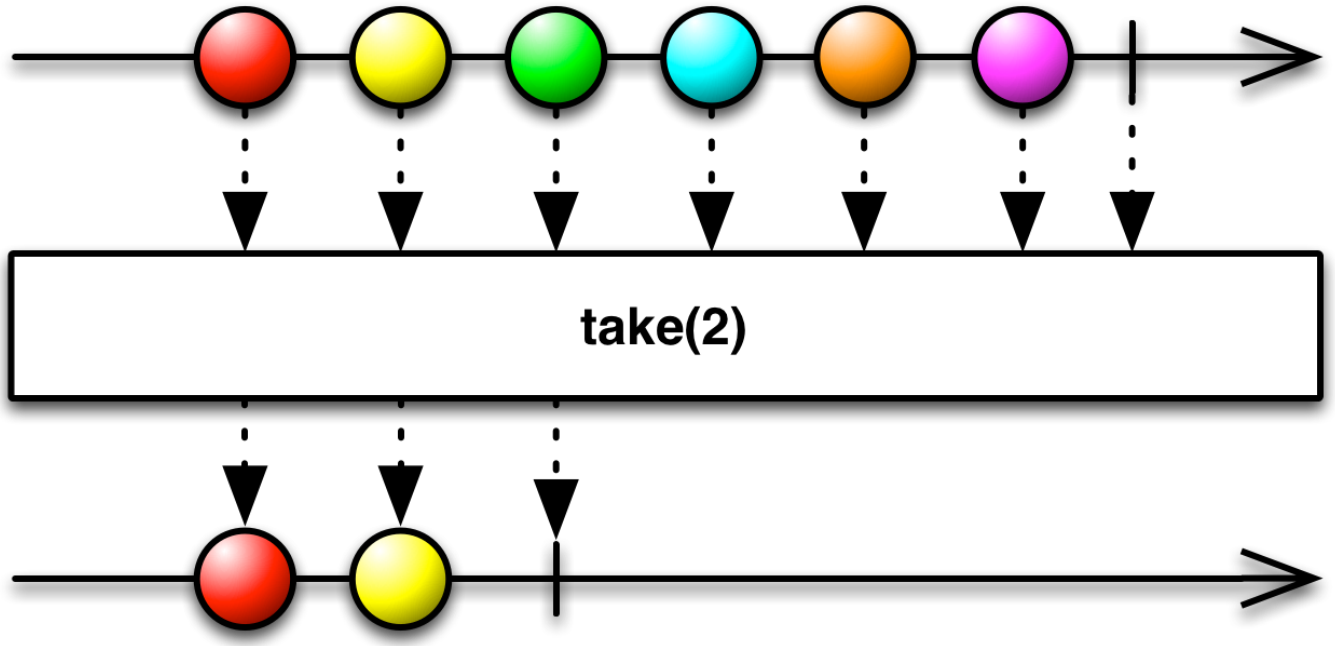
# 2 Transform


map {
}

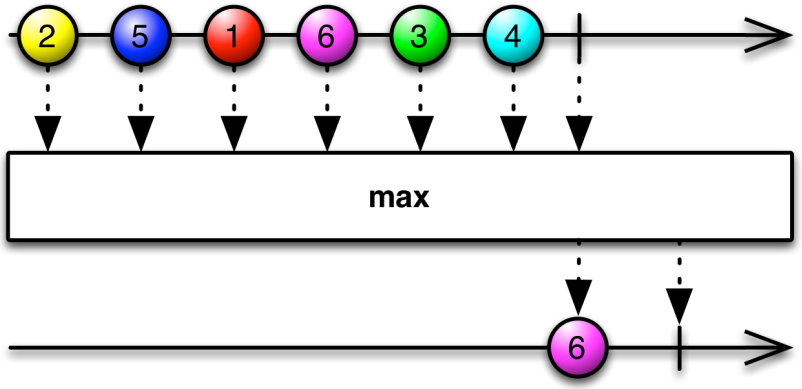

map {
}
flatten

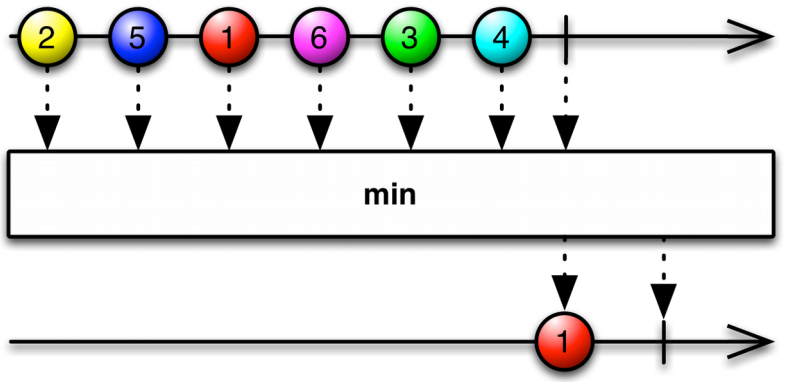
# 3 Filter

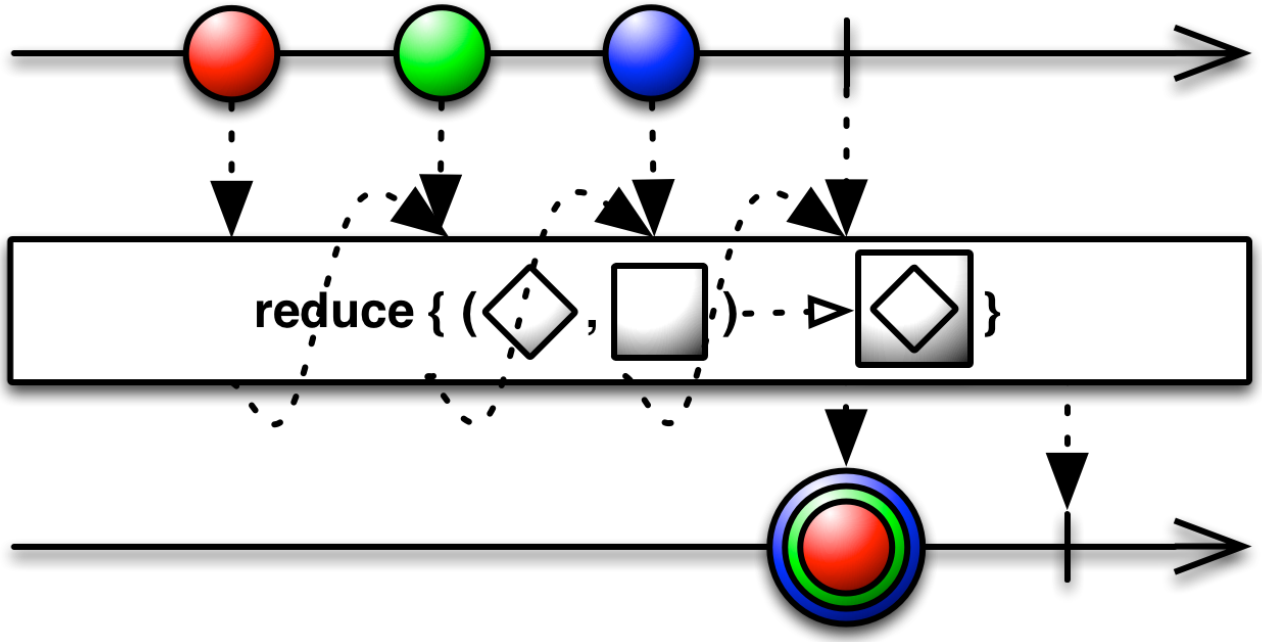

filter { }


take(2)

# 4 Aggregate


2
5
1
6
3
4
max
6


2
5
1
6
3
4
min
1

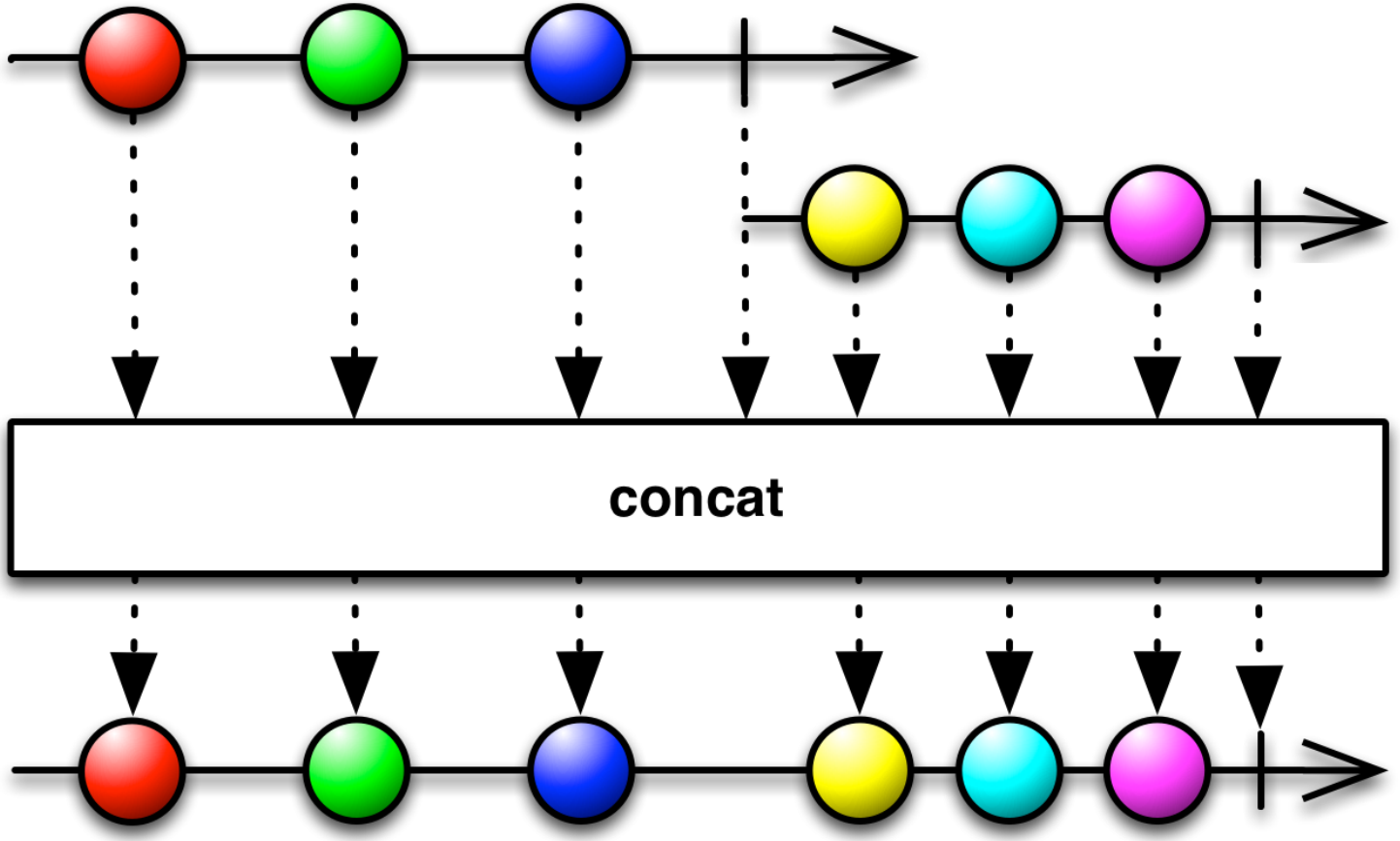

reduce { ( , )- - - }

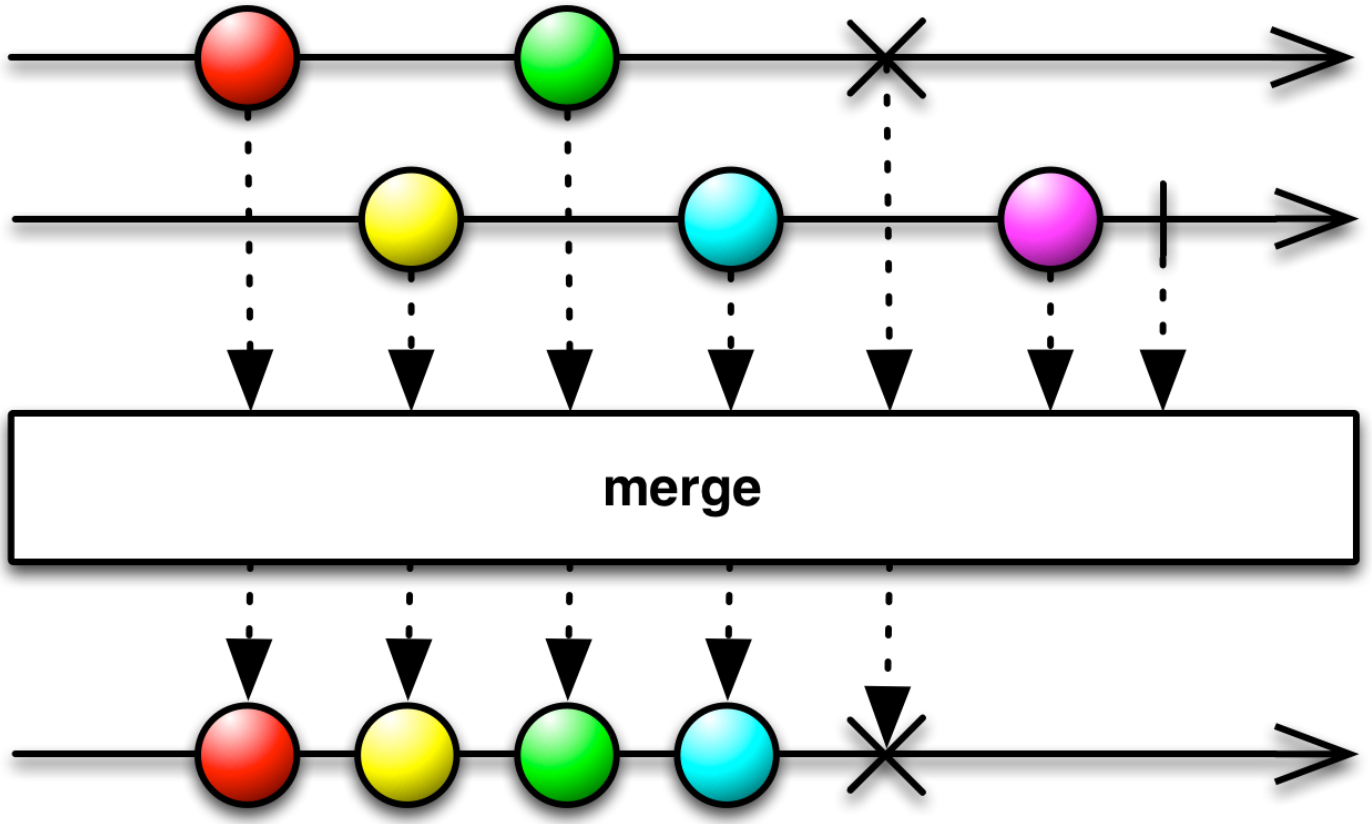
# 5 Side effects

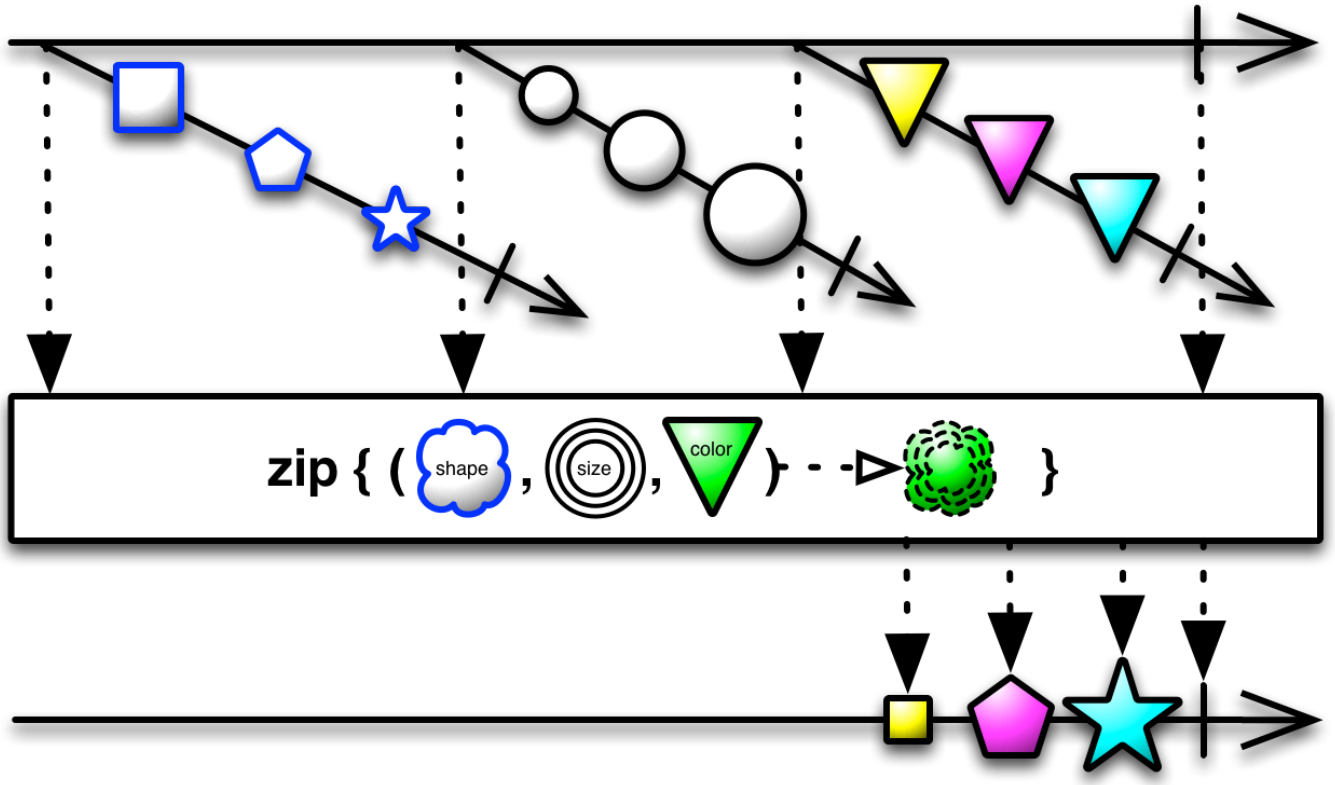
# doOnXX

- doOnNext
- doOnError
- doOnCompleted
- doOnEach

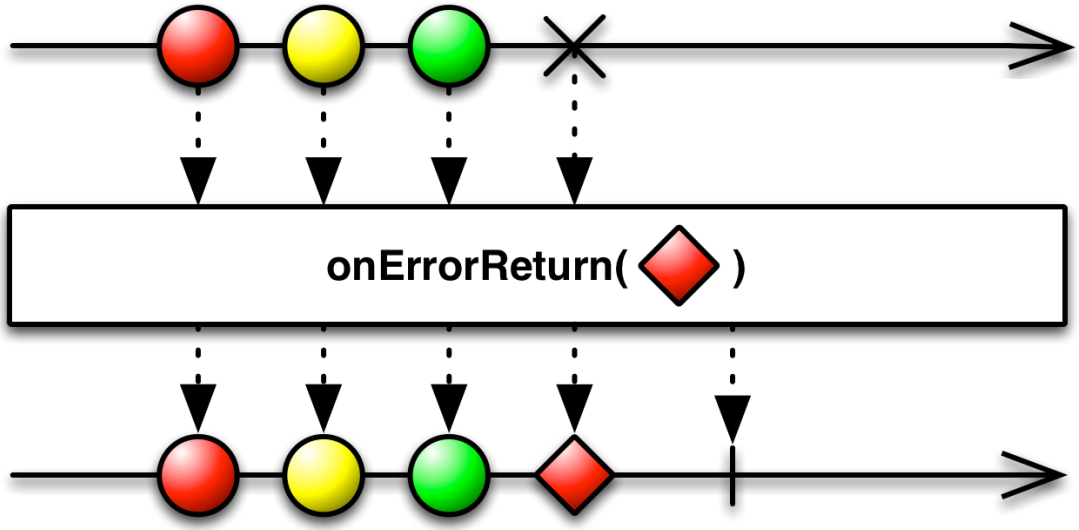
# 6 Combine

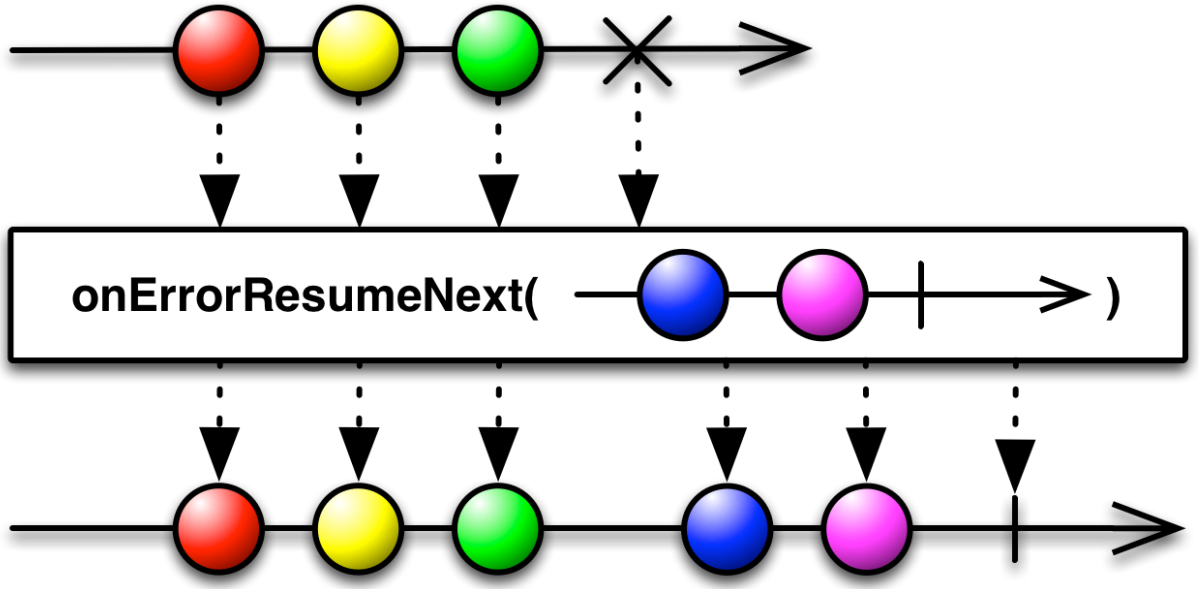

concat

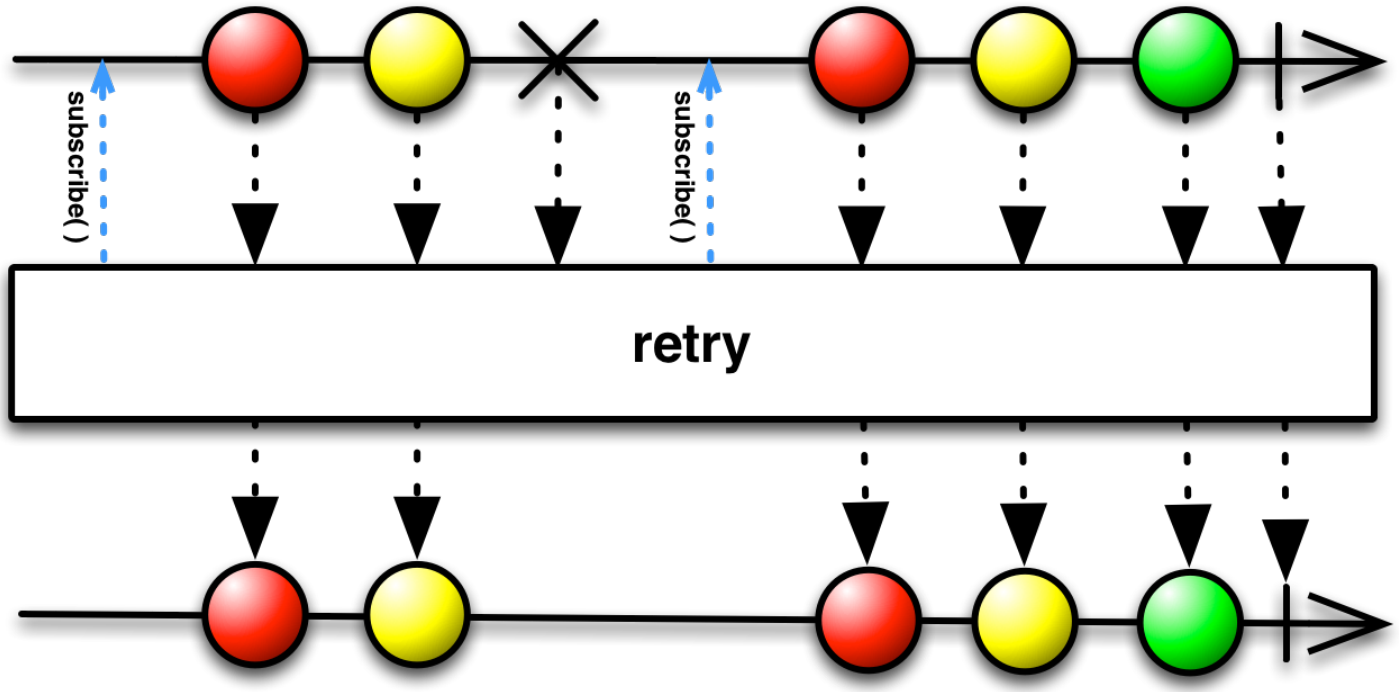

merge


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

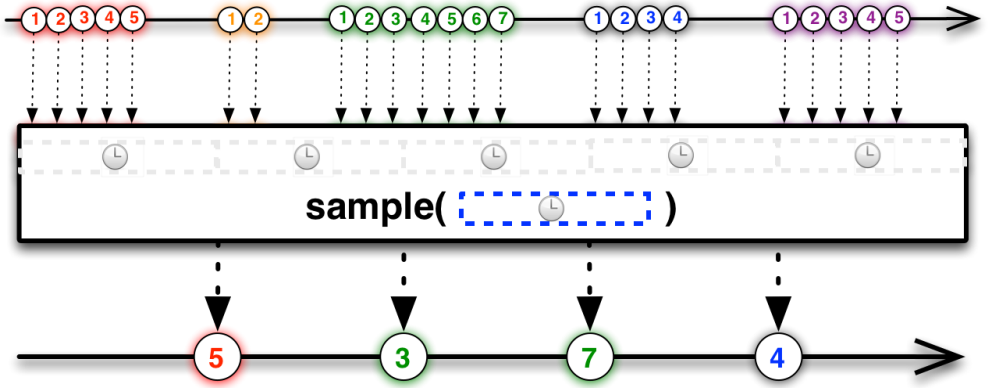
# 7 Recover / Retry


onErrorReturn( )

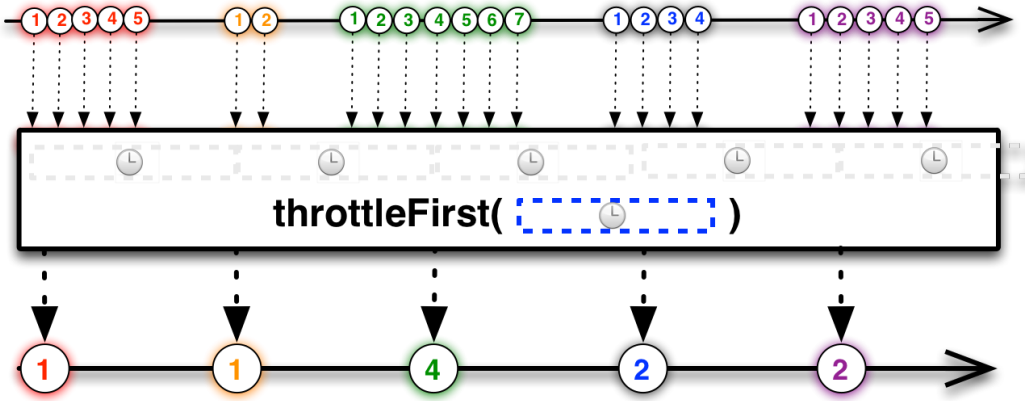

onErrorResumeNext( )

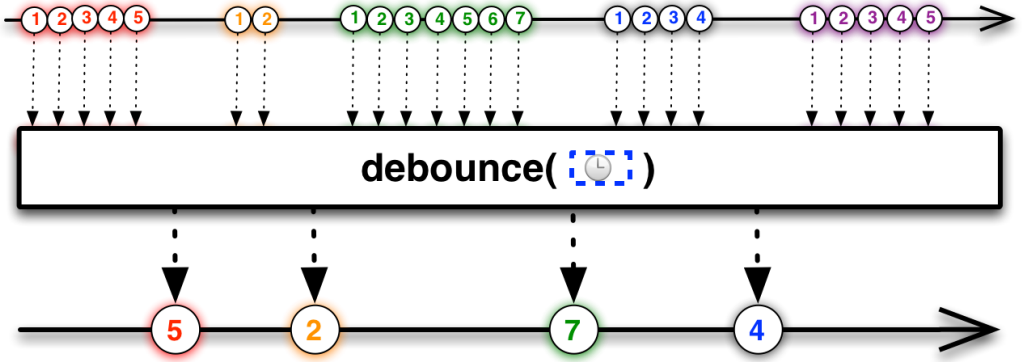

subscribe( )
subscribe( )
retry

# 8 Backpressure

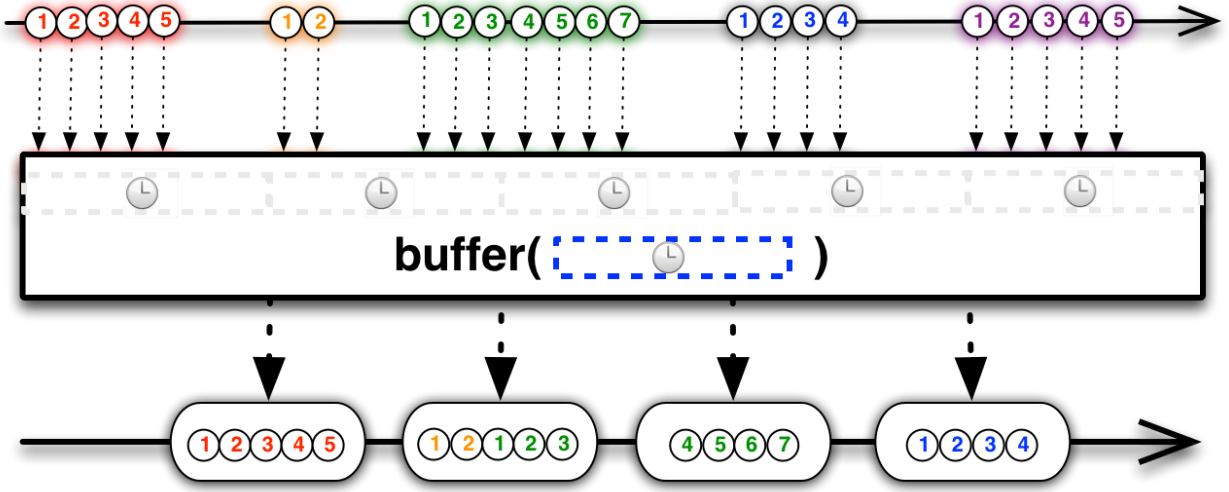


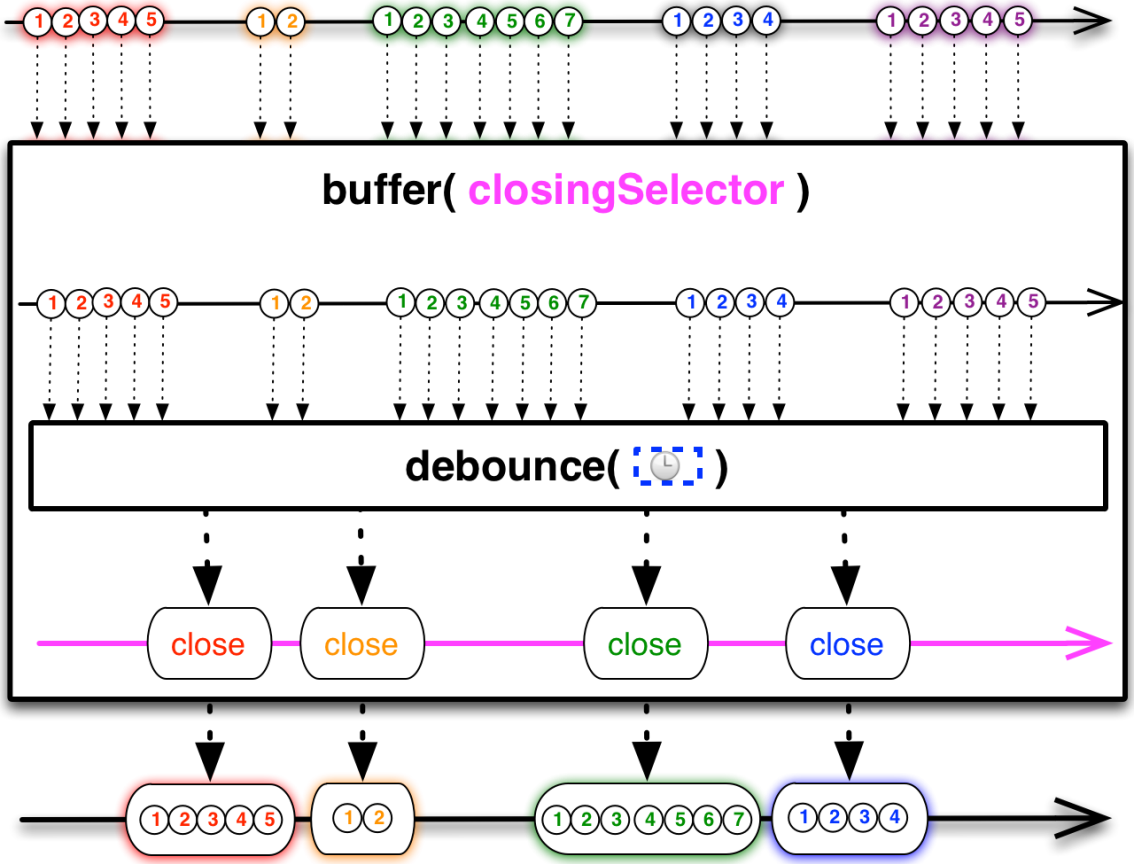

(or throttleLast)



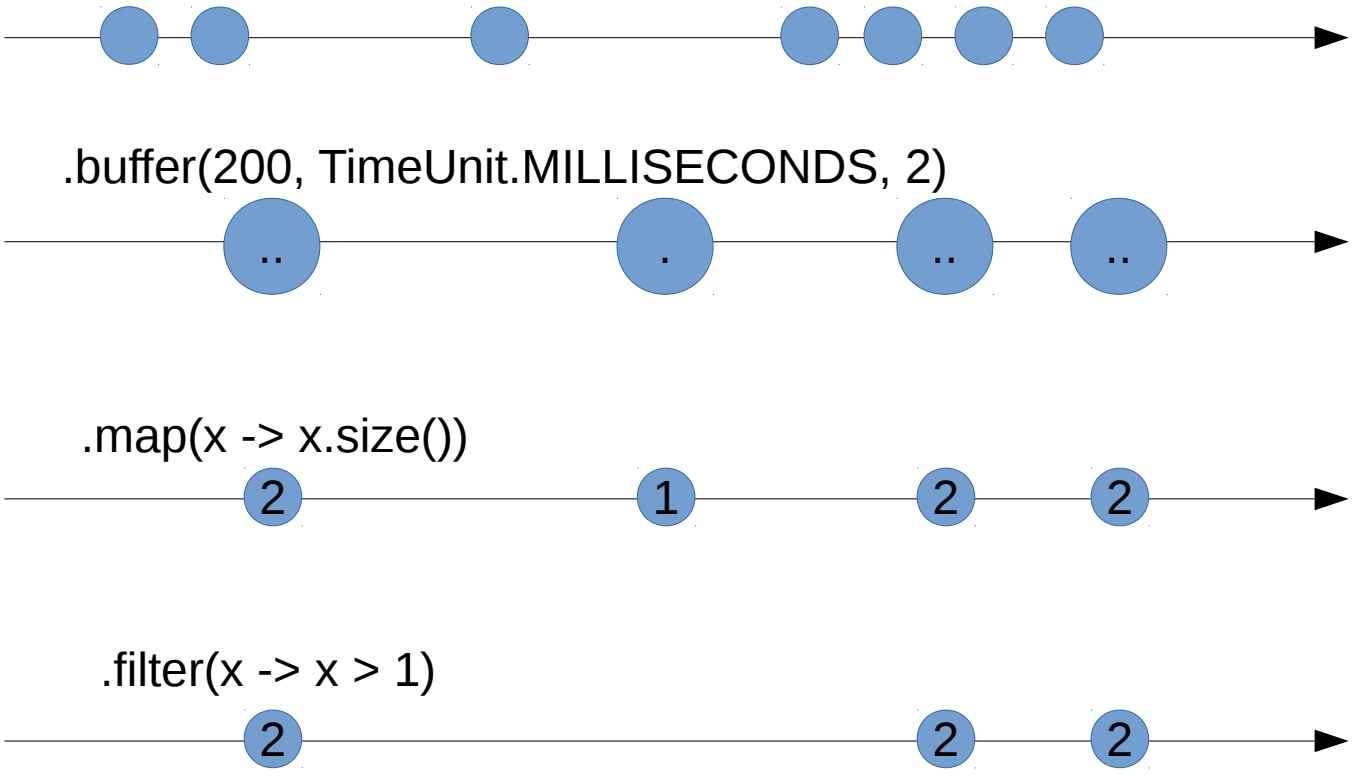




(or throttleWithTimeout)


buffer( )


buffer( closingSelector )
debounce( )
close
close
close
close

# Mouse Clicks



# 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() ->

Thank You