J8 Stream
RxJava

JavaOne™

Comparison:
patterns & performances

@JosePaumard

Why should we be interested in the Streams API?

It’s all about data processing

Up to 2014: only one tool, the Collection Framework

- Also 3rd party API: Common Collections, …

It’s all about data processing

Up to 2014: only one tool, the Collection Framework

- Also 3rd party API: Common Collections, …

From 2014: so many new APIs

It’s all about data processing

Why so?

Because data processing is more and more important

It’s all about data processing

Why so?

Because data processing is more and more important

And more and more complex!

It’s all about data processing

Why so?

Because data processing is more and more important

And more and more complex!

- Bigger and bigger amount of data to process

It’s all about data processing

Why so?

Because data processing is more and more important

And more and more complex!

- Bigger and bigger amount of data to process
- Controlled response time

It's all about data processing

Why so?

Because data processing is more and more important

And more and more complex!

- Bigger and bigger amount of data to process
- Controlled response time
- Complex algorithm

It’s all about data processing

So data processing needs high level primitives

It’s all about data processing

So data processing needs high level primitives

Able to access data wherever it is

It’s all about data processing

So data processing needs high level primitives

- Able to access data wherever it is
- That provides map / filter / reduce functions

It’s all about data processing

So data processing needs high level primitives

- Able to access data wherever it is
- That provides map / filter / reduce functions
- And efficient implementations of those!

It’s all about data processing

So data processing needs high level primitives

Able to access data wherever it is

That provides map / filter / reduce functions

And efficient implementations of those!

Most probably implemented in parallel

Agenda

1) To present two API: the Java 8 Stream API and RxJava
   - Fundamentals
   - Implemented functionalities
   - Patterns!

Agenda

1) To present two API: the Java 8 Stream API and RxJava
   - Fundamentals
   - Implemented functionalities
   - Patterns!
2) And to compare those APIs
   - From the developer point of view
   - Performances!

José PAUMARD

MCF Un. Paris 13

PhD App μ
C.S.

Open source dév.

Independant

@JosePaumard

José PAUMARD

Paris JUG

pluralsight
hardcore dev and IT training

Java magazine
By and for the Java community

Parleys

Microsoft Virtual Academy

@JosePaumard

Questions?

#J8Stream

Java 8
Stream API

API Stream

What is a Java 8 Stream?

- An object that connects to a source

API Stream

What is a Java 8 Stream?

- An object that connects to a source
- That does not hold any data

API Stream

What is a Java 8 Stream?

- An object that connects to a source
- That does not hold any data
- That implements the map / filter / reduce pattern

API Stream

What is a Java 8 Stream?

- An object that connects to a source
- That does not hold any data
- That implements the map / filter / reduce pattern

A new concept in JDK 8

API Stream

Example

```
List<String> list = Arrays.asList("one", "two", "three") ;

list.stream()
     .map(s -> s.toUpperCase())
     .max(Comparator.comparing(s -> s.length()))
     .ifPresent(s -> System.out.println(s)) ;
```

API Stream

Example

```
List<String> list = Arrays.asList("one", "two", "three") ;

list.stream() // creation of a new Stream object
    .map(s -> s.toUpperCase())
    .max(Comparator.comparing(s -> s.length()))
    .ifPresent(s -> System.out.println(s)) ;
```

API Stream

Example

```
List<String> list = Arrays.asList("one", "two", "three") ;

list.stream()
     .map(s -> s.toUpperCase()) // to upper case
     .max(Comparator.comparing(s -> s.length()))
     .ifPresent(s -> System.out.println(s)) ;
```

API Stream

Example

```
List<String> list = Arrays.asList("one", "two", "three") ;

list.stream()
     .map(s -> s.toUpperCase())
     .max(Comparator.comparing(s -> s.length())) // take the longest s
     .ifPresent(s -> System.out.println(s)) ;
```

API Stream

Example

```
List<String> list = Arrays.asList("one", "two", "three") ;

list.stream()
     .map(s -> s.toUpperCase())
     .max(Comparator.comparing(s -> s.length()))
     .ifPresent(s -> System.out.println(s)) ; // and print the result
```

API Stream

Example

```
List<String> list = Arrays.asList("one", "two", "three") ;

list.stream()
    .map(String::toUpperCase)
    .max(Comparator.comparing(String::length))
    .ifPresent(System.out::println) ; // and print the result
```

Collectors

We can use collectors

```
List<Person> list = ... ;


list.stream()
    .filter(person -> person.getAge() > 30)
    .collect(
      Collectors.groupingBy(
         Person::getAge,        // key extractor
         Collectors.counting() // downstream collector
      )
    ) ;
```

Collectors

We can use collectors

```
List<Person> list = ... ;

Map<
list.stream()
     .filter(person -> person.getAge() > 30)
     .collect(
       Collectors.groupingBy(
          Person::getAge,         // key extractor
          Collectors.counting() // downstream collector
       )
     ) ;
```

Collectors

We can use collectors

```
List<Person> list = ... ;

Map<Integer,
list.stream()
     .filter(person -> person.getAge() > 30)
     .collect(
       Collectors.groupingBy(
          Person::getAge,        // key extractor
          Collectors.counting() // downstream collector
       )
     ) ;
```

Collectors

We can use collectors

```
List<Person> list = ... ;

Map<Integer, Long> map =
list.stream()
     .filter(person -> person.getAge() > 30)
     .collect(
       Collectors.groupingBy(
          Person::getAge,         // key extractor
          Collectors.counting() // downstream collector
       )
     ) ;
```

Collectors

And we can go parallel!

```
List<Person> list = ... ;

Map<Integer, Long> map =
list.stream().parallel()
     .filter(person -> person.getAge() > 30)
     .collect(
       Collectors.groupingBy(
          Person::getAge,        // key extractor
          Collectors.counting() // downstream collector
       )
     ) ;
```

Sources of a Stream

Many ways of connecting a Stream to a source of data

Sources of a Stream

First patterns to create a Stream

```
List<Person> people = Arrays.asList(p1, p2, p3);
```

Sources of a Stream

First patterns to create a Stream

```
List<Person> people = Arrays.asList(p1, p2, p3);

Stream<Person> stream = people.stream();
```

Sources of a Stream

First patterns to create a Stream

```
List<Person> people = Arrays.asList(p1, p2, p3);

Stream<Person> stream = people.stream();

Stream<Person> stream = Stream.of(p1, p2, p3);
```

More patterns

Stream on String

```
String crazy = "supercalifragilisticexpialidocious";

IntStream letters = crazy.chars();

long count = letters.distinct().count();
```

More patterns

Stream on String

```
String crazy = "supercalifragilisticexpialidocious";

IntStream letters = crazy.chars();

long count = letters.distinct().count();
```

```
> count = 15
```

More patterns

Stream on String

```
String crazy = "supercalifragilisticexpialidocious";

IntStream letters = crazy.chars();



letters.boxed().collect(
   Collectors.groupingBy(
      Function.identity(),
      Collectors.counting()
   )
);
```

More patterns

Stream on String

```
String crazy = "supercalifragilisticexpialidocious";

IntStream letters = crazy.chars();

Map<
letters.boxed().collect(
    Collectors.groupingBy(
        Function.identity(),
        Collectors.counting()
    )
);
```

More patterns

Stream on String

```
String crazy = "supercalifragilisticexpialidocious";

IntStream letters = crazy.chars();

Map<Integer,
letters.boxed().collect(
    Collectors.groupingBy(
        Function.identity(),
        Collectors.counting()
    )
);
```

More patterns

Stream on String

```
String crazy = "supercalifragilisticexpialidocious";

IntStream letters = crazy.chars();

Map<Integer, Long> map =
letters.boxed().collect(
    Collectors.groupingBy(
        Function.identity(),
        Collectors.counting()
    )
);
```

More patterns

Stream built on the lines of a file

```
String book = "alice-in-wonderland.txt";

Stream<String> lines = Files.lines(Paths.get(book)); // autocloseable
```

More patterns

Stream built on the lines of a file

```
String book = "alice-in-wonderland.txt";

Stream<String> lines = Files.lines(Paths.get(book)); // autocloseable
```

Stream built the words of a String

```
String line = "Alice was beginning to get very tired of";

Stream<String> words = Pattern.compile(" ").splitAsStream(line);
```

Flatmap

How to mix both to get all the words of a book?

```
Function<String, Stream<String>> splitToWords =
   line -> Pattern.compile(" ").splitAsStream(line);

Stream<String> lines = Files.lines(path);

Stream<String> words = lines.flatMap(splitToWords);
```

Flatmap

Analyzing the result

```
Stream<String> words = lines.flatMap(splitToWords);

long count =
words.filter(word -> word.length() > 2)
      .map(String::toLowerCase)
      .distinct()
      .count();
```

Flatmap

Another analysis of the result

```
Stream<String> words = lines.flatMap(splitToWords);

Map<Integer, Long> map =
words.filter(word -> word.length() > 2)
      .map(String::toLowerCase)
      // .distinct()
      .collect(
         Collectors.groupingBy(
            String::length,
            Collectors.counting()
         )
      );
```

Extracting the max from a Map

Yet another analysis of the result

```
map.entrySet().stream()
    .sorted(
       Entry.<Integer, Long>comparingByValue().reversed()
    )
    .limit(3)
    .forEach(System.out::println);
```

Connecting a Stream on a source

Connecting a Stream on a non-standard source is possible

Connecting a Stream on a source

Connecting a Stream on a non-standard source is possible

A Stream is built on 2 things:

- A spliterator (comes from split and iterator)
- A ReferencePipeline (the implementation)

Connecting a Stream on a source

The Spliterator is meant to be overriden

```
public interface Spliterator<T> {

   boolean tryAdvance(Consumer<? super T> action) ;

   Spliterator<T> trySplit() ;

   long estimateSize();

   int characteristics();
}
```

Connecting a Stream on a source

The Spliterator is meant to be overriden

```
public interface Spliterator<T> {

   boolean tryAdvance(Consumer<? super T> action) ;

   Spliterator<T> trySplit(); // not needed for non-parallel processings

   long estimateSize();       // can return 0

   int characteristics();     // returns a constant
}
```

Examples of custom Spliterators

Suppose we have a Stream [1, 2, 3, …]

We want to regroup the elements: [[1, 2, 3], [4, 5, 6], …]

Let us build a GroupingSpliterator to do that

GroupingSpliterator

[1, 2, 3, 4, 5, …] -> [[1, 2, 3], [4, 5, 6], [7, 8, 9], …]

```
public class GroupingSpliterator<E> implements Spliterator<Stream<E>> {

    private final long grouping ;
    private final Spliterator<E> spliterator ;

    // implementation
}
```

```
GroupingSpliterator<Integer> gs =
    new GroupingSpliterator(spliterator, grouping);
```

GroupingSpliterator

Implementing estimateSize() and characteristics()

```
public long estimateSize() {
   return spliterator.estimateSize() / grouping ;
}
```

```
public int characteristics() {

   return this.spliterator.characteristics();
}
```

GroupingSpliterator

Implementing trySplit()

```
public Spliterator<Stream<E>> trySplit() {

   Spliterator<E> spliterator = this.spliterator.trySplit() ;
   return new GroupingSpliterator<E>(spliterator, grouping) ;
}
```

GroupingSpliterator

Implementing tryAdvance()

```
public boolean tryAdvance(Consumer<? super Stream<E>> action) {

    // should call action.accept() with the next element of the Stream

    // and return true if more elements are to be consumed

    return true ; // false when we are done
}
```

GroupingSpliterator

The structure of the resulting Stream is the following:

[[1, 2, 3], [4, 5, 6], [7, 8, 9], …]

Each element is a Stream built on the elements of the underlying Stream

GroupingSpliterator

Build a Stream element by element is done with a Stream.Builder

```
Stream.Builder<E> builder = Stream.builder() ;

for (int i = 0 ; i < grouping ; i++) {
   spliterator.tryAdvance(element -> builder.add(element);
}

Stream<E> subStream = subBuilder.build(); // [1, 2, 3]
```

GroupingSpliterator

How do we know if we are done with the underlying Stream?

```
Stream.Builder<E> builder = Stream.builder() ;

for (int i = 0 ; i < grouping ; i++) {
   spliterator.tryAdvance(
      element -> builder.add(element)
   );
}

Stream<E> subStream = subBuilder.build(); // [1, 2, 3]
```

GroupingSpliterator

How do we know if we are done with the underlying Stream?

```
Stream.Builder<E> builder = Stream.builder() ;

for (int i = 0 ; i < grouping ; i++) {
    spliterator.tryAdvance(           // when this call returns false
        element -> builder.add(element)
    );
}

Stream<E> subStream = subBuilder.build(); // [1, 2, 3]
```

GroupingSpliterator

How do we know if we are done with the underlying Stream?

```
Stream.Builder<E> builder = Stream.builder() ;
boolean finished = false;
for (int i = 0 ; i < grouping ; i++) {
   if (spliterator.tryAdvance(element -> builder.add(element)))
      finished = true;
}

Stream<E> subStream = subBuilder.build(); // [1, 2, 3]
```

GroupingSpliterator

How do we know if we are done with the underlying Stream?

```
public boolean tryAdvance(Consumer<? super Stream<E>> action) {
   Stream.Builder<E> builder = Stream.builder() ;
   boolean finished = false;
   for (int i = 0 ; i < grouping ; i++) {
      if (spliterator.tryAdvance(element -> builder.add(element)))
         finished = true;
   }

   Stream<E> subStream = subBuilder.build(); // [1, 2, 3]
   action.accept(subStream) ;
   return !finished ;
}
```

RollingSpliterator

What about building a Stream like this one:

[[1, 2, 3], [2, 3, 4], [3, 4, 5], …]

This time we need a ring buffer

RollingSpliterator

The tryAdvance() call on the underlying Stream becomes this:

```
private boolean advanceSpliterator() {
   return spliterator.tryAdvance(
      element -> {
         buffer[bufferWriteIndex.get() % buffer.length] = element ;
         bufferWriteIndex.incrementAndGet() ;
      });
}
```

bufferWriteIndex is an AtomicLong

RollingSpliterator

Building the element streams from the ring buffer:

```
private Stream<E> buildSubstream() {

    Stream.Builder<E> subBuilder = Stream.builder() ;
    for (int i = 0 ; i < grouping ; i++) {
        subBuilder.add(
            (E)buffer[(i + bufferReadIndex.get()) % buffer.length]
        ) ;
    }
    bufferReadIndex.incrementAndGet() ;
    Stream<E> subStream = subBuilder.build() ;
    return subStream ;
}
```

RollingSpliterator

Putting things together to build the tryAdvance() call

```java
public boolean tryAdvance(Consumer<? super Stream<E>> action) {
   boolean finished = false ;

   if (bufferWriteIndex.get() == bufferReadIndex.get()) {
      for (int i = 0 ; i < grouping ; i++) {
         if (!advanceSpliterator()) {
            finished = true ;
         }
      }
   }
   if (!advanceSpliterator()) {
      finished = true ;
   }

   Stream<E> subStream = buildSubstream() ;
   action.accept(subStream) ;
   return !finished ;
}
```

Spliterator on Spliterator

We saw how to build a Stream on another Stream by rearranging its elements

What about building a Stream by merging the elements of other Streams?

Spliterator on Spliterators

Let us take two Streams:

[1, 2, 3, 4, …]

[a, b, c, d, …]

And build a ZippingSpliterator:

[F(1, a), F(2, b), F(3, c), …]

Spliterator on Spliterators

What about

estimateSize()

trySplit()

characteristics()?

They are the same as the underlying streams

Spliterator on Spliterators

We then need to implement tryAdvance()

```
public boolean tryAdvance(Consumer<? super R> action) {
   return spliterator1.tryAdvance(
      e1 -> {
         spliterator2.tryAdvance(e2 -> {
            action.accept(tranform.apply(e1, e2)) ;
         }) ;
      }) ;
}
```

Where *transform* is a BiFunction

ZippingSpliterator

What about creating a Builder for this Spliterator?

```
ZippingSpliterator.Builder<String, String, String> builder =
   new ZippingSpliterator.Builder();

ZippingSpliterator<String,String,String> zippingSpliterator = builder
   .with(spliterator1)
   .and(spliterator2)
   .mergedBy((e1, e2) -> e1 + " - " + e2)
   .build();
```

ZippingSpliterator

The complete pattern:

```
Stream<String>  stream1 = Stream.of("one", "two", "three");
Stream<Integer> stream2 = Stream.of(1, 2, 3);

Spliterator<String>  spliterator1 = stream1.spliterator();
Spliterator<Integer> spliterator2 = stream2.spliterator();



Stream<String> zipped =
   StreamSupport.stream(zippingSpliterator, false);
```

What did we do with Streams so far?

We took one stream and built a stream by regrouping its elements in some ways

What did we do with Streams so far?

We took one stream and built a stream by regrouping its elements in some ways

We took to streams and we merged them, element by element, using a bifunction

What did we do with Streams so far?

We took one stream and built a stream by regrouping its elements in some ways

We took to streams and we merged them, element by element, using a bifunction

What about taking one element at a time, from different streams?

The WeavingSpliterator

We have N streams, and we want to build a stream that takes:

- 1st element from the 1st stream
- 2nd element from the 2nd stream, etc…

The WeavingSpliterator

The estimateSize():

```
public long estimateSize() {
   int size = 0 ;
   for (Spliterator<E> spliterator : this.spliterators) {
      size += spliterator.estimateSize() ;
   }
   return size ;
}
```

The WeavingSpliterator

The tryAdvance():

```
private AtomicInteger whichOne = new AtomicInteger();

public boolean tryAdvance(Consumer<? super E> action) {

   return spliterators[whichOne.getAndIncrement() %
                       spliterators.length]
             .tryAdvance(action);
}
```

What did we do with Streams so far?

We took one stream and built a stream by regrouping its elements in some ways

We took to streams and we merged them, element by element, using a bifunction

We took a collection of streams and built a stream by taking elements from them, in a given order

Wrap-up for the Stream API

The Java 8 Stream API is not just about implementing map / filter / reduce of building hashmaps

We can use it to manipulate data in advanced ways:

- by deciding on what source we want to connect
- by deciding how we can consume the data from that source

Reactive Streams
RxJava

RxJava

Open source API, available on Github

Developed by Netflix

RxJava is the Java version of ReactiveX

- .NET
- Python, Kotlin, JavaScript, Scala, Ruby, Groovy, Rust
- Android

https://github.com/ReactiveX/RxJava

RxJava

RxJava is not an alternative implementation of Java 8 Streams, nor the Collection framework

It is an implementation of the Reactor pattern

RxJava

The central class is the Observable class

RxJava

The central class is the Observable class

It’s big: ~10k lines of code

RxJava

The central class is the Observable class

It’s big: ~10k lines of code

It’s complex: ~100 static methods, ~150 non-static methods

RxJava

The central class is the Observable class

It's big: ~10k lines of code

It's complex: ~100 static methods, ~150 non-static methods

It's complex: not only because there is a lot of things in it, but also because the concepts are complex

RxJava

Interface Observer

used to « observe » an observable

RxJava

Interface Observer

used to « observe » an observable

Interface Subscription

used to model the link between an observer and an observable

Interface Observer

A simple interface

```
public interface Observer<T> {

   public void onNext(T t);

   public void onCompleted();

   public void onError(Throwable e);
}
```

How to subscribe

Subscribing to an observable

```
Observable<T> observable = ... ;

Subscription subscription = observable.subscribe(observer) ;
```

How to subscribe

Subscribing to an observable

```
Observable<T> observable = ... ;

Subscription subscription = observable.subscribe(observer) ;
```

```
public interface Subscription {

   public void unsubscribe();

   public void isUnsubscribe();
}
```

RxJava – agenda

Patterns to create Observables

How to merge observables together

Hot observables / backpressure

The usual ways

Observable from collections and arrays

```
Observable<String> obs1 = Observable.just("one",  "two", "three") ;

List<String> strings = Arrays.asList("one",  "two", "three") ;
Observable<String> obs2 = Observable.from(strings) ;
```

The usual ways

Observable useful for tests

```
Observable<String> empty = Observable.empty() ;
Observable<String> never = Observable.never() ;
Observable<String> error = Observable.<String>error(exception) ;
```

The usual ways

Series and time series

```
Observable<Long> longs = Observable.range(1L, 100L) ;

// interval
Observable<Long> timeSerie1 =
   Observable.interval(1L, TimeUnit.MILLISECONDS) ; // a serie of longs

// initial delay, then interval
Observable<Long> timeSerie2 =
   Observable.timer(10L, 1L, TimeUnit.MILLISECONDS) ; // one 0
```

The usual ways

The using() method

```
public final static <T, Resource> Observable<T> using(
   final Func0<Resource> resourceFactory,                       // producer
   final Func1<Resource, Observable<T>> observableFactory, // function
   final Action1<? super Resource> disposeAction           // consumer
) { }
```

The usual ways

The using() method

```
public final static <T, Resource> Observable<T> using(
   final Func0<Resource> resourceFactory,                      // producer
   final Func1<Resource, Observable<T>> observableFactory, // function
   final Action1<? super Resource> disposeAction             // consumer
) { }
```

How using() works

1) a resource is created using the resourceFactory
2) an observable is created from the resource using the observableFactory
3) this observable is further decorated with actions for exception handling

RxJava & executors

Some of those methods take a further argument

```
Observable<Long> longs = Observable.range(0L, 100L, scheduler) ;
```

Scheduler is an interface (in fact an abstract class, to define « default methods » in Java 7)

Schedulers should be used to create schedulers

Schedulers

Factory Schedulers

```
public final class Schedulers {

   public static Scheduler immediate() {...}   // immediate

   public static Scheduler newThread() {...}   // new thread

   public static Scheduler trampoline() {...} // queued in the current
                                              // thread
}
```

Schedulers

Factory Schedulers

```
public final class Schedulers {

    public static Scheduler computation() {...} // computation ES

    public static Scheduler io() {...}          // IO growing ES

    public static Scheduler test() {...}

    public static Scheduler from(Executor executor) {...}
}
```

Schedulers

Schedulers can be seen as Executors

Some of them are specialized (IO, Computation)

Observers are called in the thread that runs the Observable

Schedulers

The Schedulers.from(executor) is useful to call observers in certain threads

For instance:

```
Scheduler swingScheduler =
    Schedulers.from(
        SwingUtilities::invokeLater
    );
```

A 1st example

A simple example

```
Observable<Integer> range1To100 = Observable.range(1L, 100L) ;
range1To100.subscribe(System.out::println) ;
```

A 1st example

A simple example

```
Observable<Integer> range1To100 = Observable.range(1L, 100L) ;
range1To100.subscribe(System.out::println) ;
```

```
> 1 2 3 4 ... 100
```

A 2nd example

A not so simple example

```
Observable<Integer> timer = Observable.timer(1, TimeUnit.SECONDS) ;
timer.subscribe(System.out::println) ;
```

A 2nd example

A not so simple example

```
Observable<Integer> timer = Observable.timer(1, TimeUnit.SECONDS) ;
timer.subscribe(System.out::println) ;
```

```
>
```

Nothing is printed

A 2nd example

Let us modify this code

```
Observable<Integer> timer = Observable.timer(1, TimeUnit.SECONDS) ;
timer.subscribe(() -> {
   System.out.println(Thread.currentThread().getName() + " " +
                      Thread.currentThread().isDaemon()) ;
}) ;
Thread.sleep(2) ;
```

A 2nd example

Let us modify this code

```
Observable<Integer> timer = Observable.timer(1, TimeUnit.SECONDS) ;
timer.subscribe(() -> {
   System.out.println(Thread.currentThread().getName() + " " +
                      Thread.currentThread().isDaemon()) ;
}) ;
Thread.sleep(2) ;
```

```
> RxComputationThreadPool-1 - true
```

About the 1st & 2nd examples

The first example ran in the main thread

About the 1st & 2nd examples

The first example ran in the main thread

The second example ran in a daemon thread, that does not prevent the JVM from exiting. Nothing was printed out, because it did not have the time to be executed.

About the 1st & 2nd examples

The first example ran in the main thread

The second example ran in a daemon thread, that does not prevent the JVM from exiting. Nothing was printed out, because it did not have the time to be executed.

Observable are ran in their own schedulers (executors)

Merging Observable together

RxJava provides a collection of methods to merge observables together

Merging Observable together

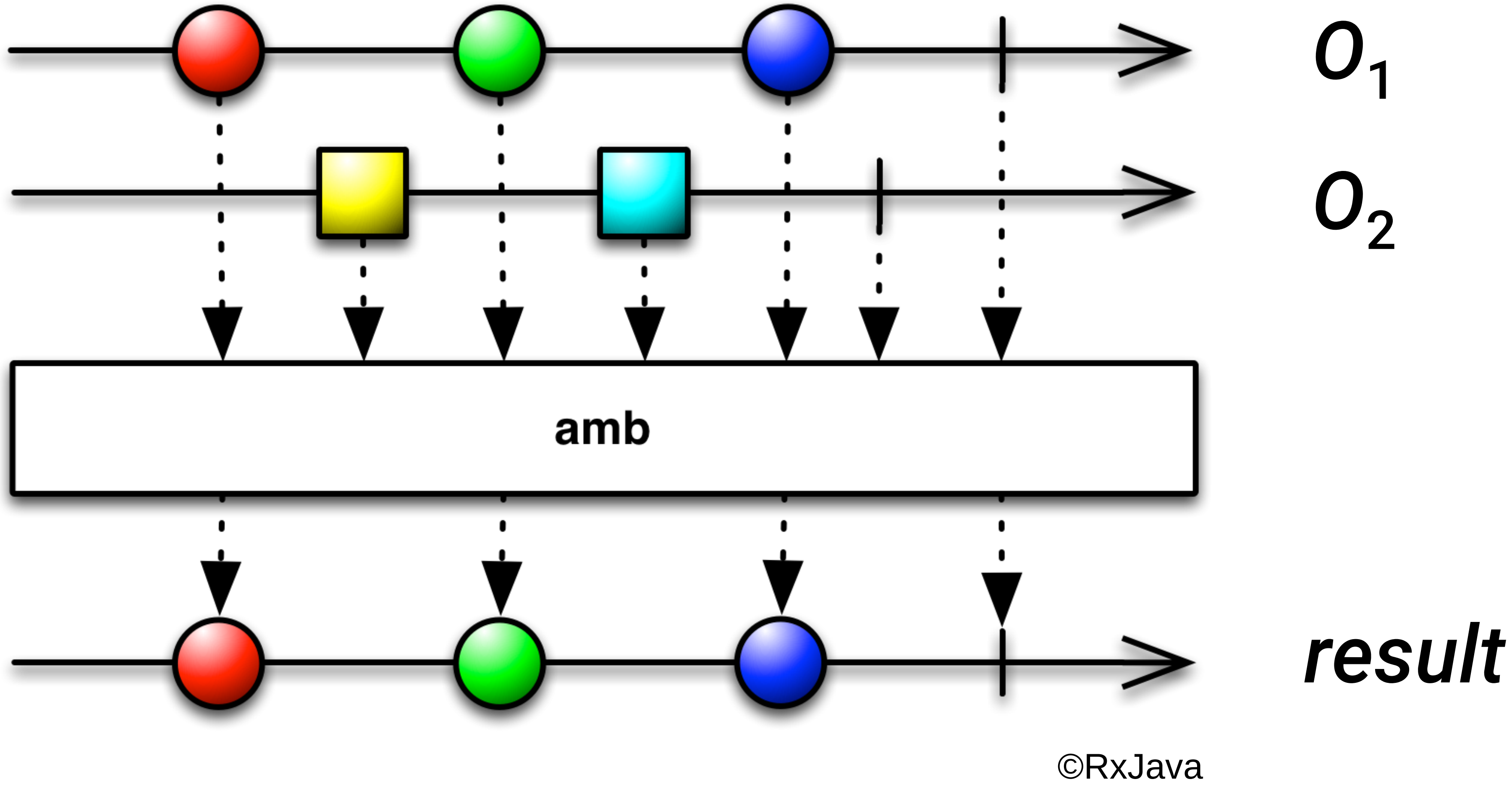
RxJava provides a collection of methods to merge observables together

With many different, very interesting semantics

The ambiguous operator

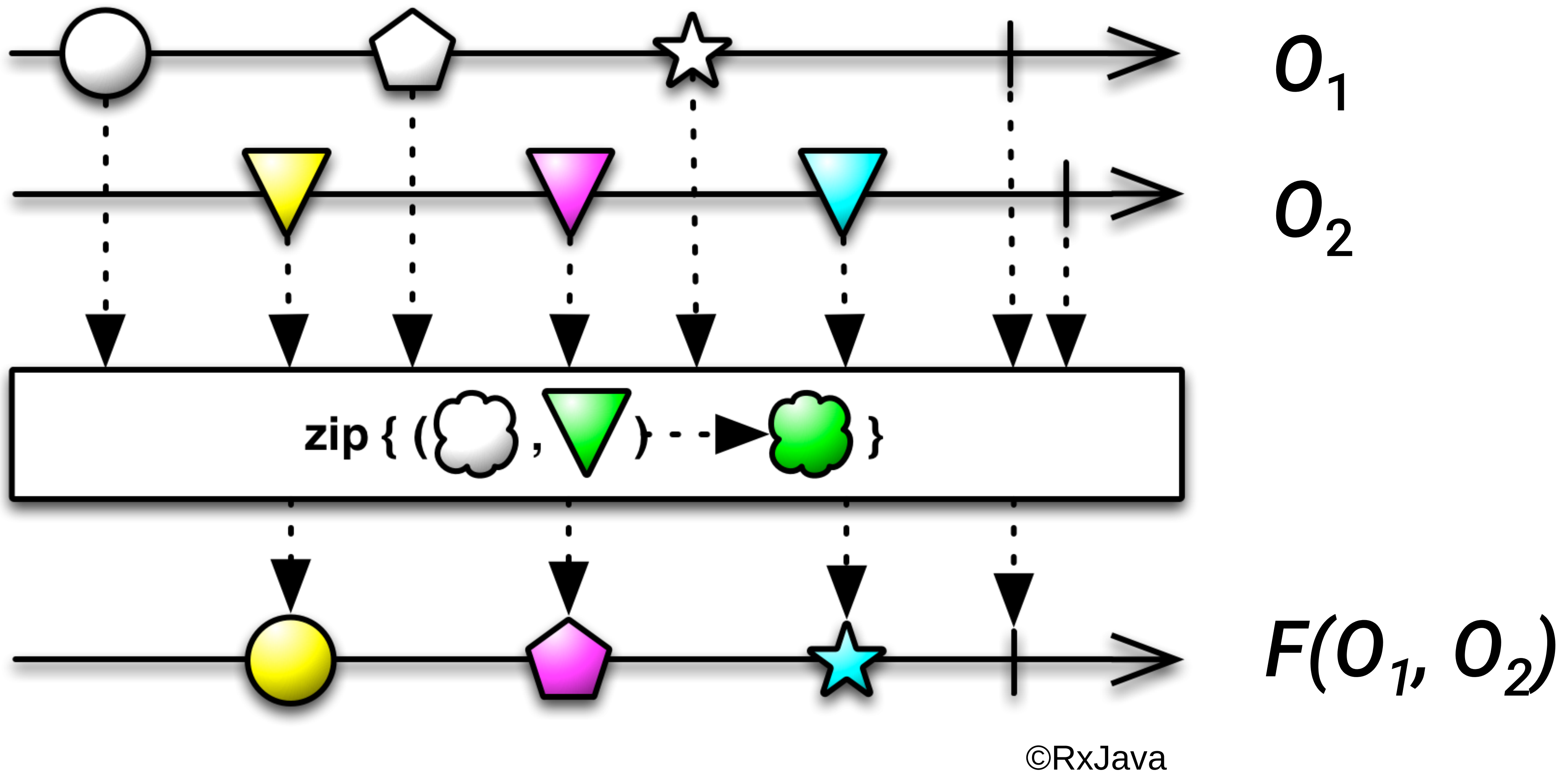
The first operator is amb(), that stands for « ambiguous »

It takes the first Observable that produced data



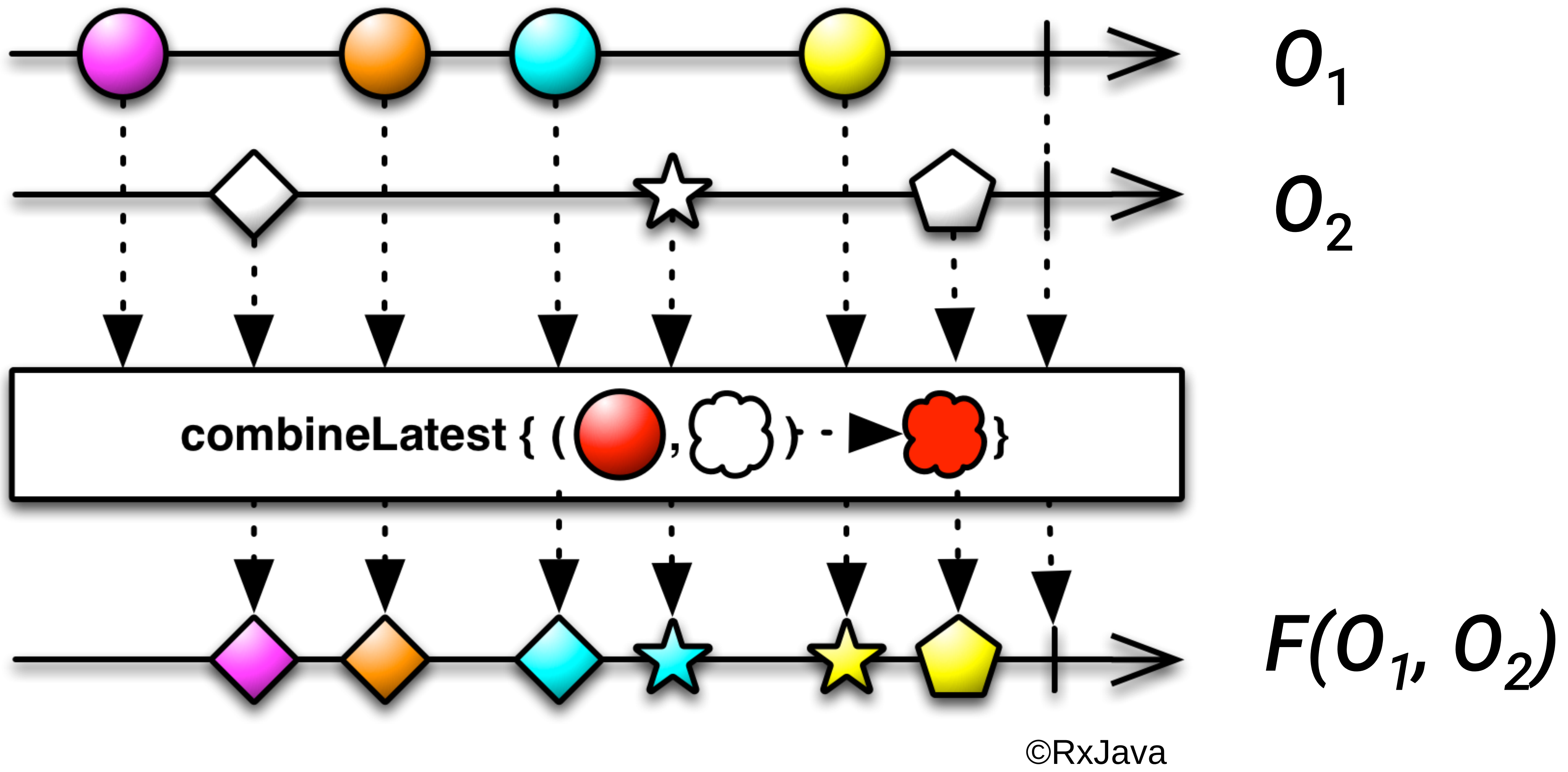
The zip operator

The zip operator takes one element from each Observable and combine them using a function



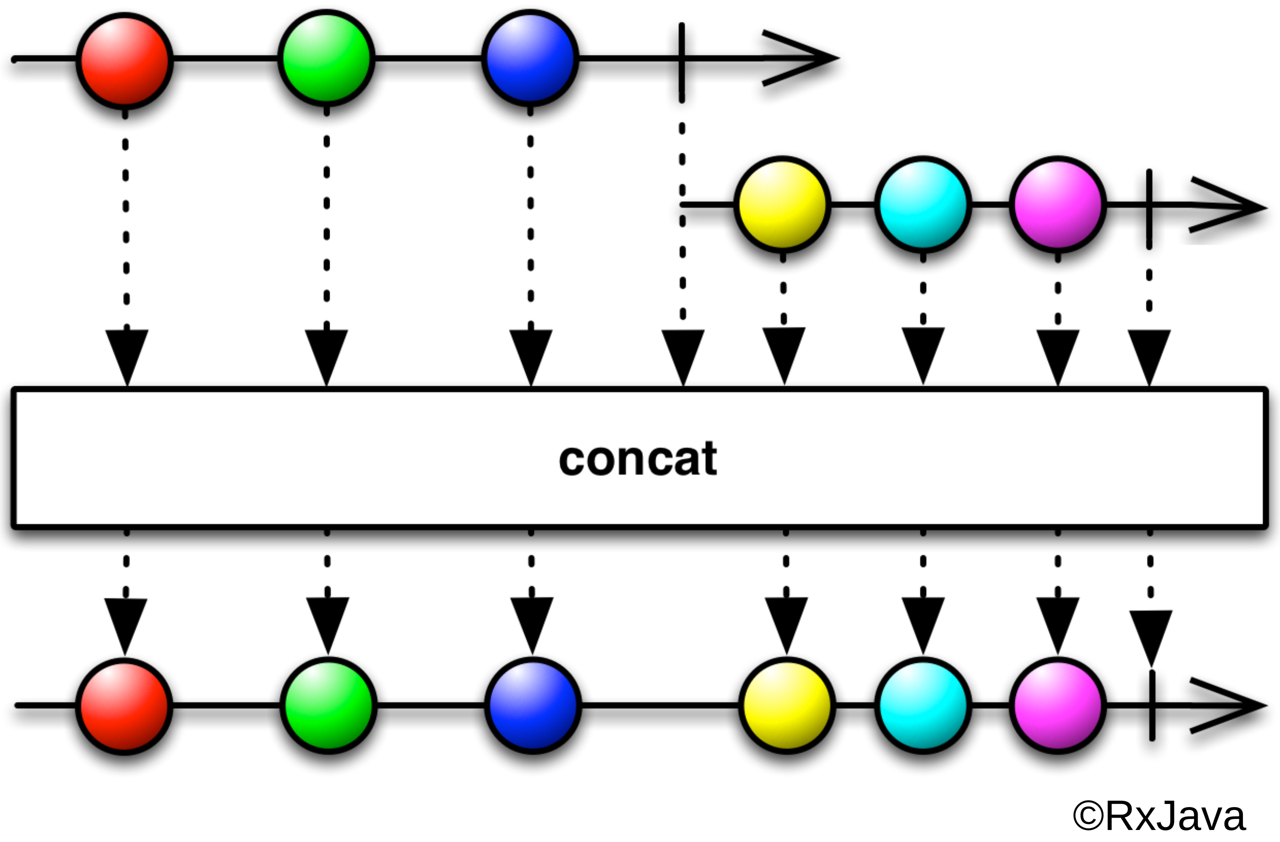
The combine latest operator

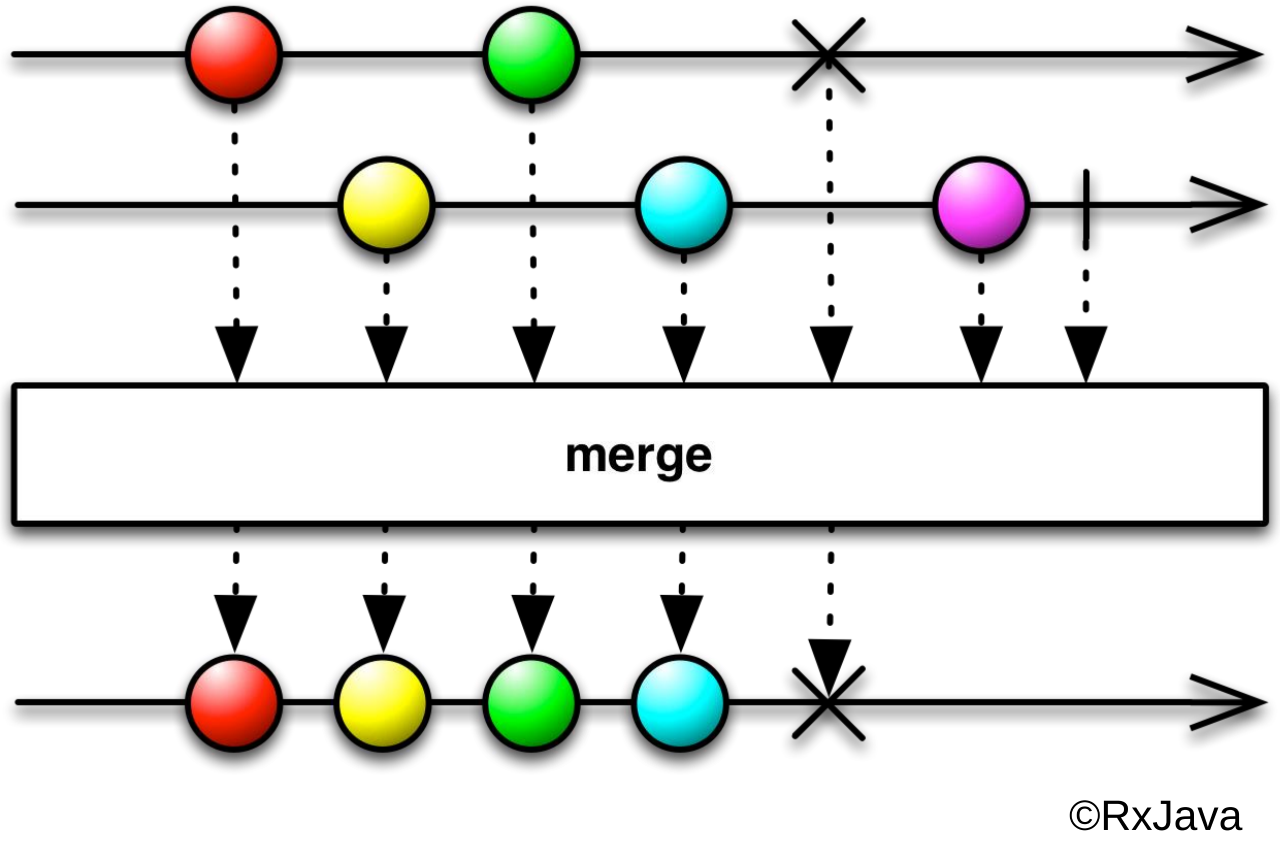
The combineLatest is a non-strict zip operator, emits an item each time an observable emits one



Concat and merge

Concat: emits O_1 and then O_2, without mixing them, merge stops on the emission of an error





Observables of Observables

The methods we saw are defined on Iterables of Observables

```
public final static <T> Observable<T> merge(
   Iterable<Observable<T>> listOfSequences) { }
```

Observables of Observables

The methods we saw are defined on Iterables of Observables

```
public final static <T> Observable<T> merge(
   Iterable<Observable<T>> listOfSequences) { }
```

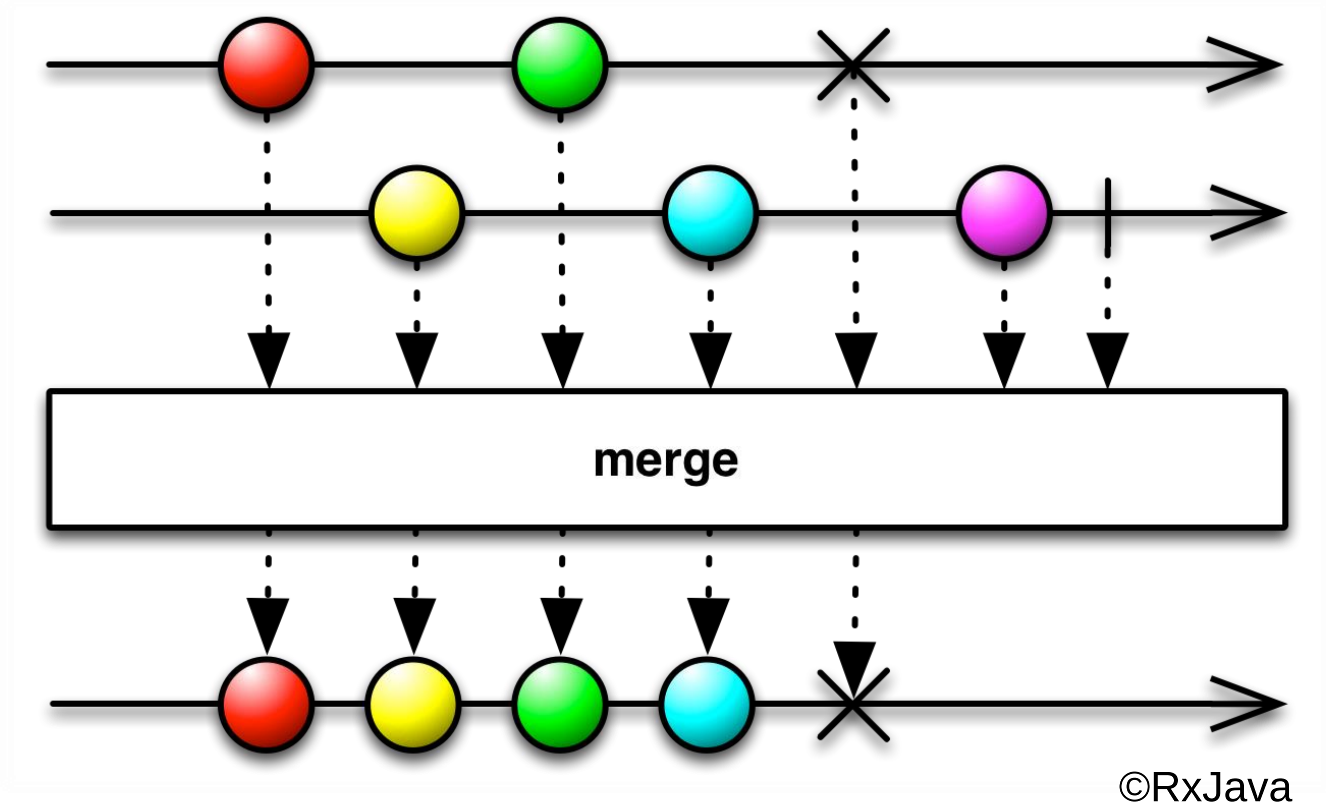
They are also defined on Observables of Observables

```
public final static <T> Observable<T> merge(
   Observable<Observable<T>> sequenceOfSequences) { }
```

Observables of Observables

And the marble diagrams are different

```
public final static <T> Observable<T> merge(
   Iterable<Observable<T>> listOfSequences) { }
```

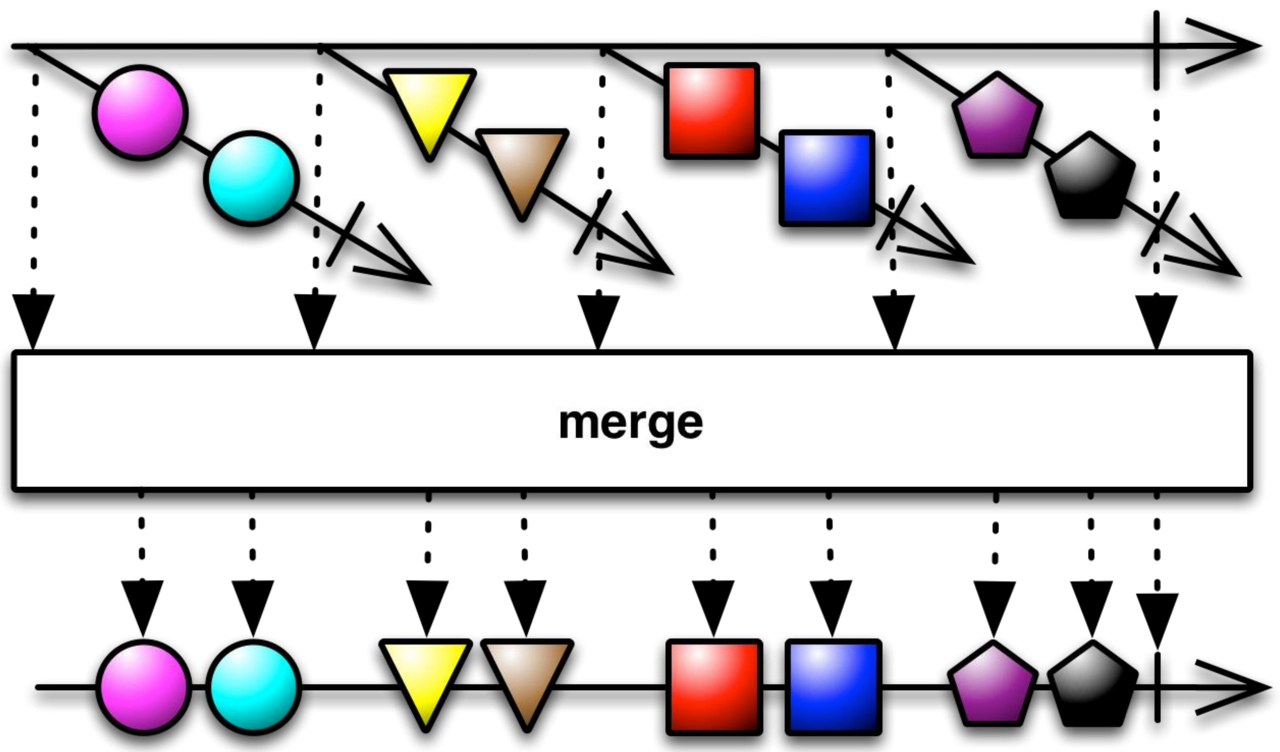


©RxJava

Observables of Observables

And the marble diagrams are different

```
public final static <T> Observable<T> merge(
   Observable<Observable<T>> sequenceOfSequences) { }
```

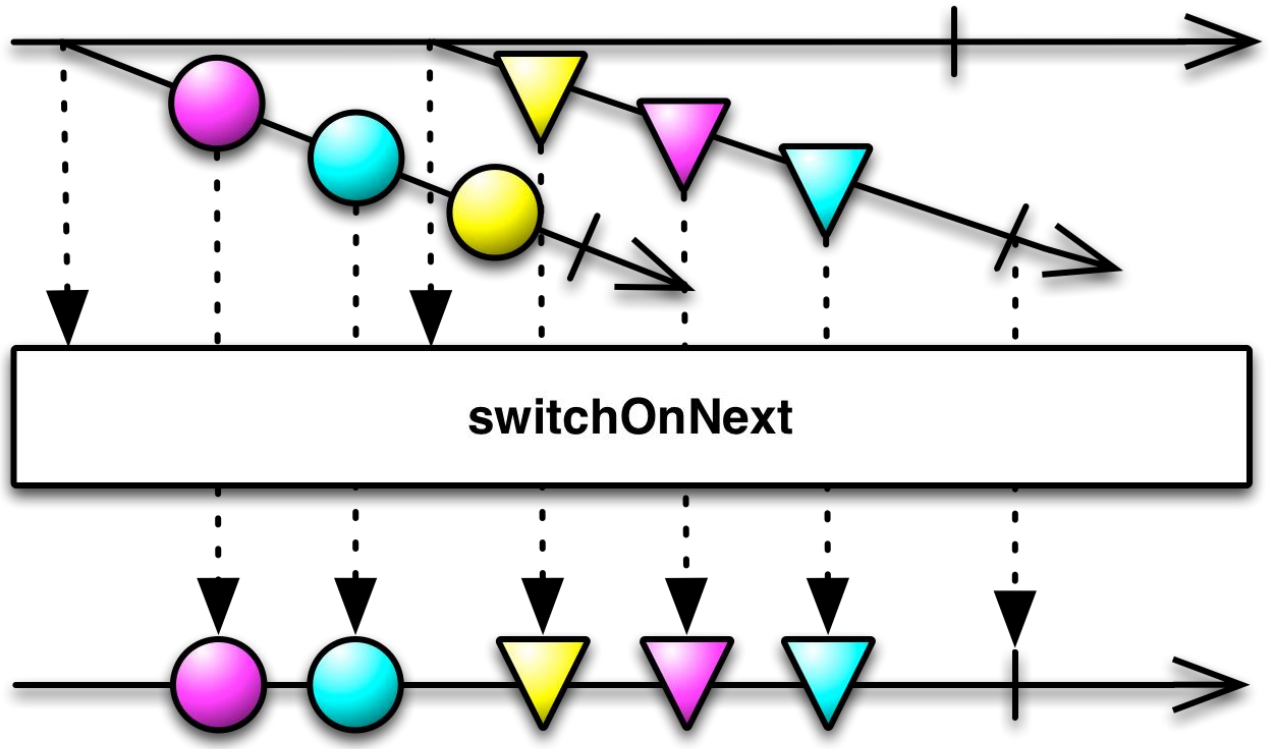


©RxJava

Observables of Observables

Then comes the switchOnNext operator

```
public final static <T> Observable<T> switchOnNext(
   Observable<Observable<T>> sequenceOfSequences) { }
```



©RxJava

A 3rd example

A little more tricky

```
Observable<Integer> range = Observable.range(1, 100) ;

Observable<String> manyStrings =
Observable.combineLatest(
    range, Observable.just("one"),
    (integer, string) -> string) ;
```

A 3rd example

A little more tricky

```
Observable<Integer> range = Observable.range(1, 100) ;

Observable<String> manyStrings =
Observable.combineLatest(
    range, Observable.just("one"),
    (integer, string) -> string) ;
```

```
> one (and nothing more)
```

A 3rd example

A little more tricky

```
Observable<Integer> range = Observable.range(1, 100) ;

Observable<String> manyStrings =
Observable.combineLatest(
    range, Observable.just("one"),
    (integer, string) -> string) ;
```

```
> one (and nothing more)
```

Combines two source Observables by emitting an item that aggregates the latest values of each of the source Observables **each time an item is received from either of the source Observables,** where this aggregation is defined by a specified function.

A 4th example

Ok, so what about this one?

```
Observable<Integer> timer = Observable.interval(3, TimeUnit.SECONDS) ;

Observable<String> manyStrings =
Observable.combineLatest(
    timer, Observable.just("one"),
    (integer, string) -> string) ;
```

A 4th example

Ok, so what about this one?

```
Observable<Integer> timer = Observable.interval(3, TimeUnit.SECONDS) ;

Observable<String> manyStrings =
Observable.combineLatest(
    timer, Observable.just("one"),
    (integer, string) -> string) ;
```

```
> one one one one one one ...
```

A 4th example

Ok, so what about this one?

```
Observable<Integer> timer = Observable.interval(3, TimeUnit.SECONDS) ;

Observable<String> manyStrings =
Observable.combineLatest(
    timer, Observable.just("one"),
    (integer, string) -> string) ;
```

```
> one one one one one one ...
```

It seems that *timer* does not behave as *range*…

Cold and hot observables

And indeed it does not!

Cold and hot observables

And indeed it does not!

Range is *cold* observable = produces when observed

Cold and hot observables

And indeed it does not!

Range is *cold* observable = produces when observed

Timer is a *hot* observable = produces, observed or not

Cold and hot observables

And indeed it does not!

Range is *cold* observable = produces when observed

Timer is a *hot* observable = produces, observed or not

Be careful, a cold observable can become hot

Cold and hot observables

An observable can have as many observers as we want

Thus, a cold observable observed by an observer will generate values according to this observer

If another observer subscribes to it, it will see this observable as a hot observable

Cold and hot observables

Problem: what happens if we have many observers to subscribe?

Cold and hot observables

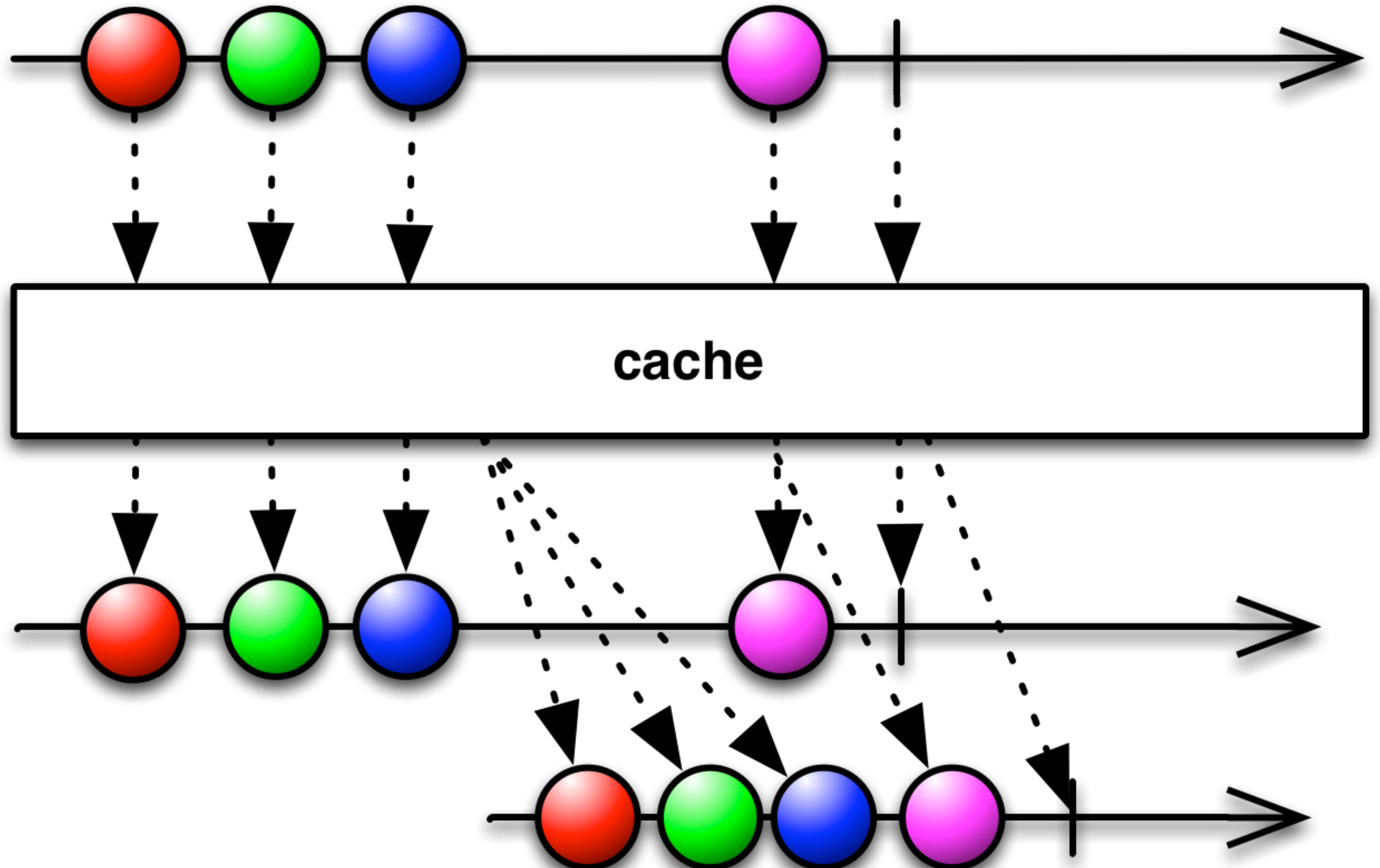
Problem: what happens if we have many observers to subscribe?

Our application could miss the first values emitted by an observable

Cold and hot observables

We can use the cache method

```
cache(int capacity) { }
```



Cold and hot observables

We can do better and use a ConnectableObservable (which is an Observable)

```
// does not count as a subscription
ConnectableObservable<Long> publish = observable.publish();


// take the time to connect all the observers

// then call
publish.connect();
```

Subscribing to an Observable

About 10 different versions of doOnSomething

```
doOnEach(Action1<T> onEach) { } // consumer
```

Each = the 3 callbacks: onNext, onError, onComplete

For instance, onNext() only acts with the onNext callback

Subscribing to an Observable

Map / filter, return an Observable

```
map(Func1<T, R> mapping) { }
cast(Class<R> clazz) { }      // casts the elements
```

Emits only the elements that match the predicate

```
filter(Func1<T, Boolean> predicate) { }
```

Subscribing to an Observable

Materialize: special type of mapping

```
materialize() { } // Observable<Notification<T>>
```

Emit a Notification object that wraps the item, with metadata

Useful for logging

Does the reverse:

```
dematerialize() { } // Observable<T>
```

Subscribing to an Observable

Selecting ranges of elements

```
first() { }
last() { }
skip(int n) { }
limit(int n) { }
take(int n) { }
```

Subscribing to an Observable

Special functions

```
exists(Func1<T, Boolean> predicate) { }
```

Emits a true then complete on the onComplete
if the Observable emitted at least one item

Subscribing to an Observable

Special functions

```
elementAt(int n) { }
```

Emits the n^{th} item then complete on the onNext
if the Observable emitted at least *n* items

Subscribing to an Observable

Special functions

```
ignoreElement() { }
```

Emits nothing then complete on the onComplete

Subscribing to an Observable

Special functions

```
single() { }
```

Emits the first item on the onComplete
or emits an error on the second item emitted

Subscribing to an Observable

Special functions

```
single(Func1<T, Boolean> predicate) { }
```

Emits the matching item if it has been seen, on onComplete or emits an error if not, on onComplete

Reductions & collections

Classical reductions

```
all(Func1<T, Boolean> predicate) { }    // Observable<Boolean>
count() { }                              // Observable<Long>
```

```
forEach(Action1<T> consumer) { }        // void
```

Reductions & collections

Accumulations

```
reduce(Func2<T, T, T> accumulator) { } // Observable<T>
```

```
scan(Func2<T, T, T> accumulator) { }   // Observable<T>
```

Reduce: returns the reduction of all the items, on onComplete

Scan: returns the reduction step by step, on onNext

Reductions & collections

Collecting data in a mutable container

```
collect(Func0<R> stateFactory,        // producer
        Action2<R, T> collector) { } // BiConsumer
```

Example: adding the elements to a list

```
collect(ArrayList::new,     // producer
        ArrayList::add) { } // BiConsumer
```

Reductions & collections

Ready to use collectors for lists:

```
toList() { }       // Observable<List<T>>
toSortedList() { } // Observable<List<T>>
```

Reductions & collections

Ready to use collectors for lists:

```
toList() { }       // Observable<List<T>>
toSortedList() { } // Observable<List<T>>
```

And for maps (toMap can lose items):

```
toMultimap(Func1<T, K> keySelector,         // Observable<
           Func1<T, V> valueSelector) { } //    Map<K, Collection<T>>

toMap(Func1<T, K> keySelector) { } // Observable<Map<K, T>>
```

Reductions & collections

A special kind of map:

```
groupBy(Func1<T, K> keySelector) { } // function
```

The returned observable could hold a single map,
but it does not

Reductions & collections

A special kind of map:

```
groupBy(Func1<T, K> keySelector) { } // function
```

The returned observable could hold a single map,
but it does not

It holds GroupedObservable items, which holds its key

Repeating & retrying

The repeat method repeats the same Observable

```
repeat() { }                // Observable<T>
repeat(long times) { }  // Observable<T>

repeatWhen(
   Func1<Observable<Void>>, Observable<?>> notificationHandler
) { }
```

Repeating & retrying

repeat: will repeat the same Observable again and again

On the returned Observable, one can invoke:

- onComplete() or onError(), which will trigger the same call on the source Observable
- onNext(), that triggers the repetition

Repeating & retrying

The retry method will reset the Observable on error

```
retry() { }                // Observable<T>
retry(long times) { }  // Observable<T>

retryWhen(
   Func1<Observable<Throwable>>, Observable<?>> notificationHandler
) { }
```

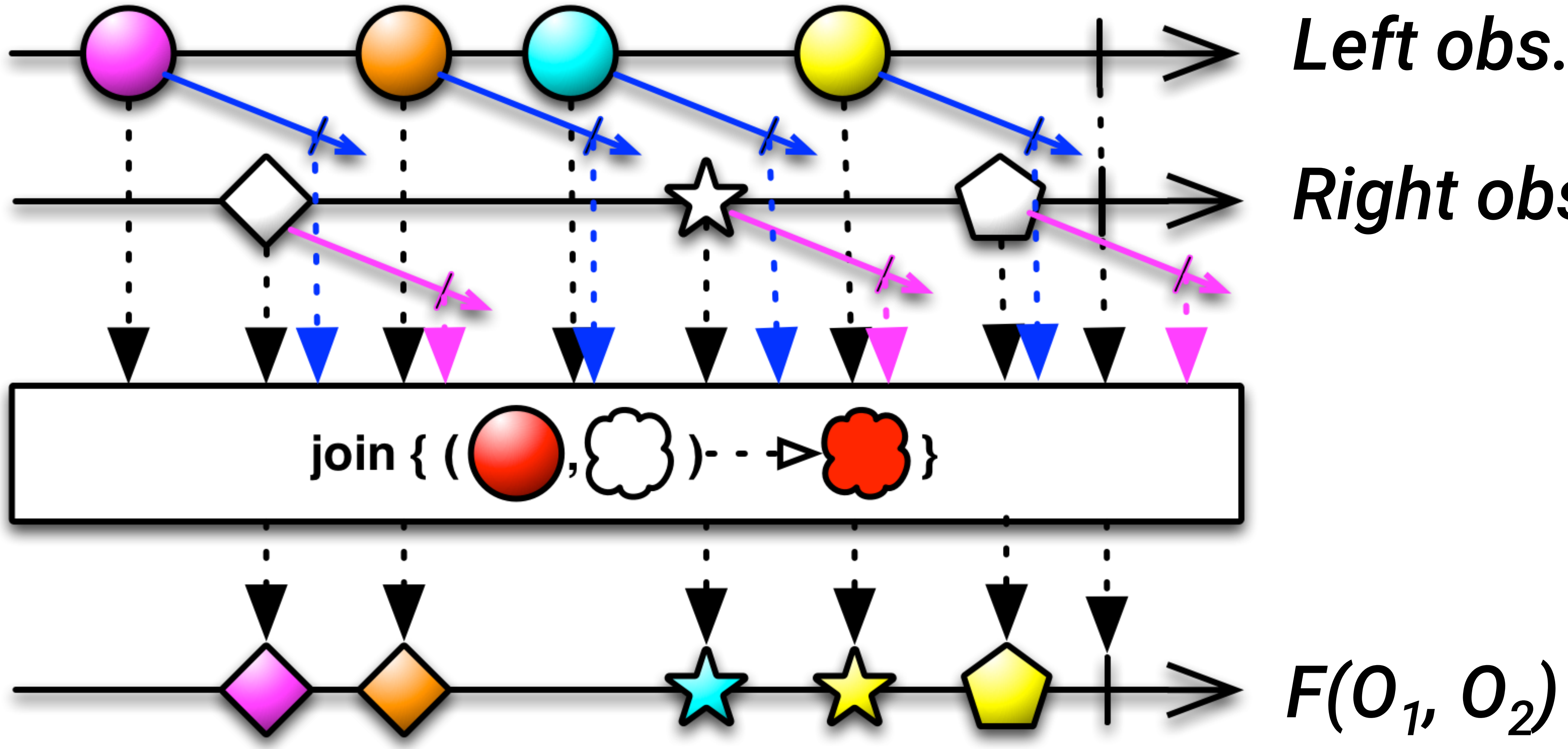
Joining

Joins to Observable, based on the overlapping of durations

```
join(Observable<TRight> right,
     Func1<T, Observable<TLeftDuration>> leftDurationSelector,
     Func1<TRight, Observable<TRightDuration>> rightDurationSelector,
     Func2<T, TRight, R> resultSelector) { }
```

Join

Joins to Observable, based on the overlapping of durations



©RxJava

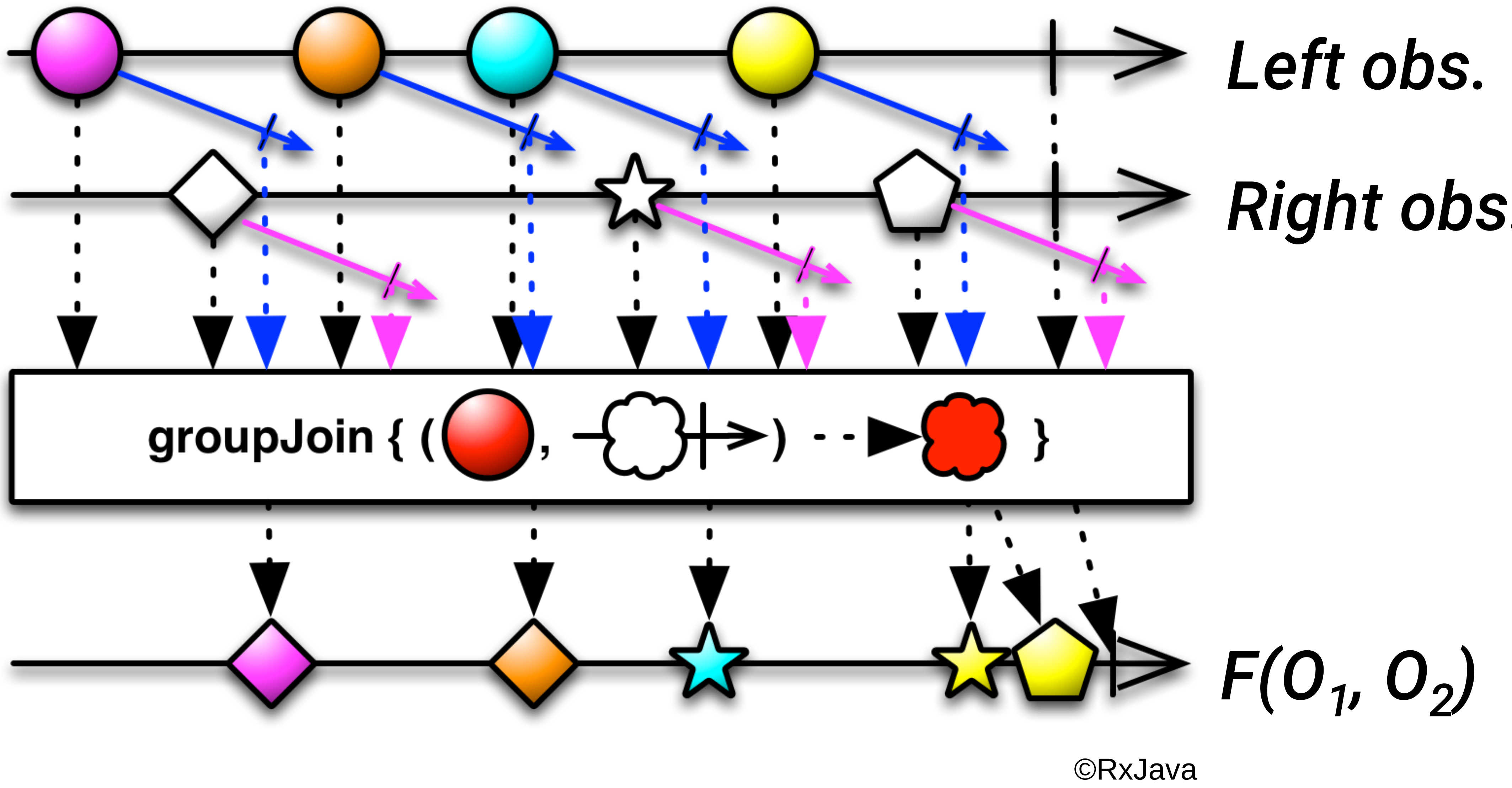
GroupJoin

Joins to Observable, based on the overlapping of durations

```
groupJoin(
  Observable<T2> right,
  Func1<? super T, ? extends Observable<D1>> leftDuration,
  Func1<? super T2, ? extends Observable<D2>> rightDuration,
  Func2<? super T, ? super Observable<T2>, ? extends R> resultSelector)
{ }
```

GroupJoin

Joins to Observable, based on the overlapping of durations



Dealing with the time

RxJava has a set of methods to deal with time

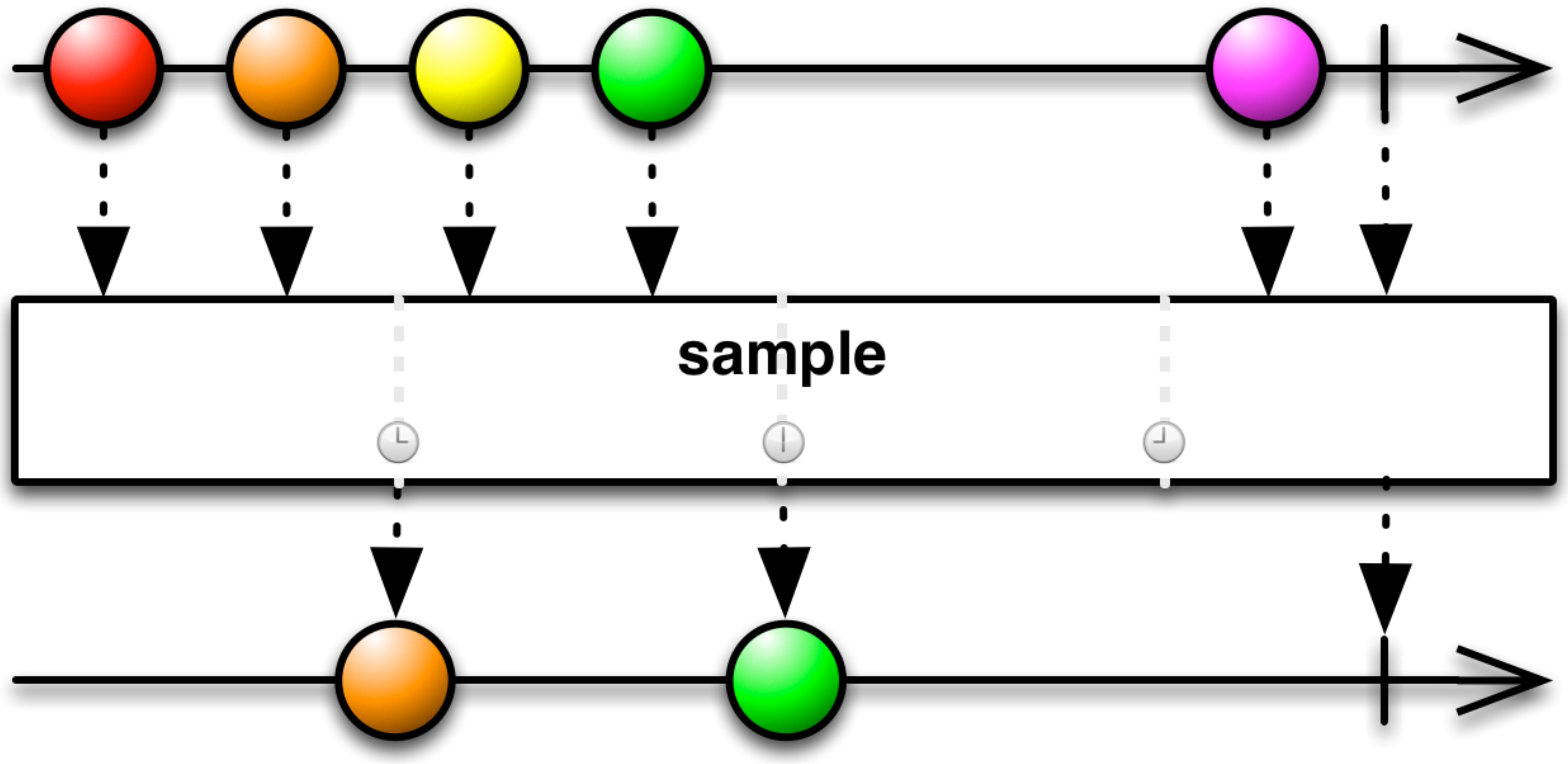
Let us go back on our cold / hot Observables

It is easy to imagine a hot Observable that emits an onNext event each second, thus playing the role of a clock

Sampling

With hot observables, RxJava can synchronize on a clock

```
sample(long period, TimeUnit timeUnit) { }
```

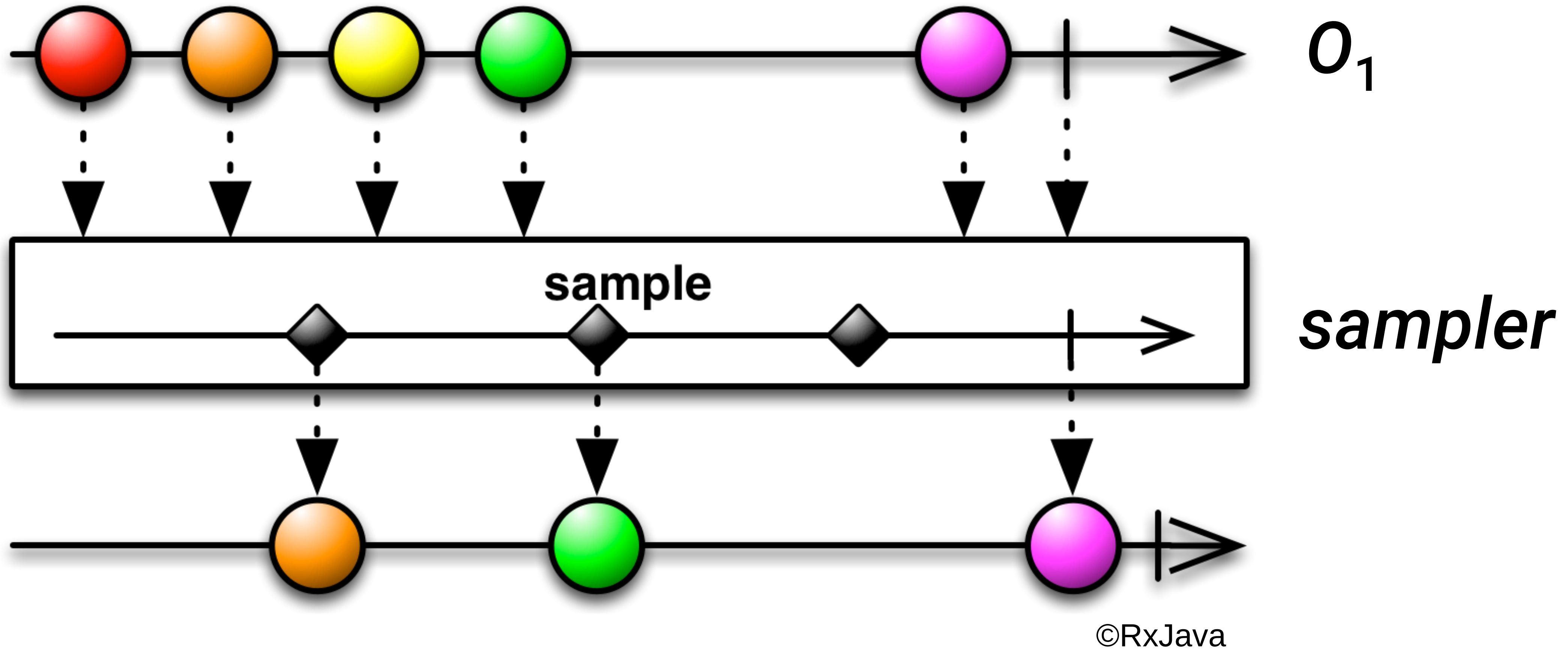


©RxJava

Sampling

Or synchronize on a reference Observable!

```
sample(Observable<U> sampler) { } // samples on emission & completion
```



RxJava and synchronization

A very powerful feature:

RxJava can synchronize on a clock (real-time)

And on another Observable, it can then take its own time scale

Measuring time

If we can measure time, then we can emit it

```
timeInterval() { } // Observable<TimeInterval<T>>
```

Will emit an the amount of time between the two last onNext events

Measuring time

If we can measure time, then we can delay an emission

```
delay(long delay, TimeUnit timeUnit) ; // Observable<T>
```

Will reemit the items, with a delay

This delay can be computed from the items themselves

```
delay(Func1<T, Observable<U> func1) ; // Observable<T>
```

Measuring time

If we can measure time, then we can timeout

```
timeout(long n, TimeUnit timeUnit) { }
```

Will emit an error if no item is seen during this time

Fast hot observables

What happens if a hot Observable emits too many items?

What happens when an Observer cannot keep up the pace of an Observable?

Fast hot observables

What happens if a hot Observable emits too many items?

What happens when an Observer cannot keep up the pace of an Observable?

This leads to the notion of *backpressure*

Backpressure methods

Backpressure is a way to slow down the emission of elements

It can act on the observing side

Several strategies:

- Buffering items
- Skipping items (sampling is a way to implement this)

Buffering

Buffering records data in a buffer and emits it

```
buffer(int size) { }                               // Observable<List<T>>

buffer(long timeSpan, TimeUnit unit) { } // Observable<List<T>>
buffer(long timeSpan, TimeUnit unit, int maxSize) { }
```

Buffering

Buffering records data in a buffer (a list) and emits it

```
buffer(int size) { }                             // Observable<List<T>>

buffer(long timeSpan, TimeUnit unit) { } // Observable<List<T>>
buffer(long timeSpan, TimeUnit unit, int maxSize) { }

buffer(Observable<O> bufferOpenings,                       // Openings events
       Func1<O, Observable<C>> bufferClosings) { } // Closings events
```

Windowing

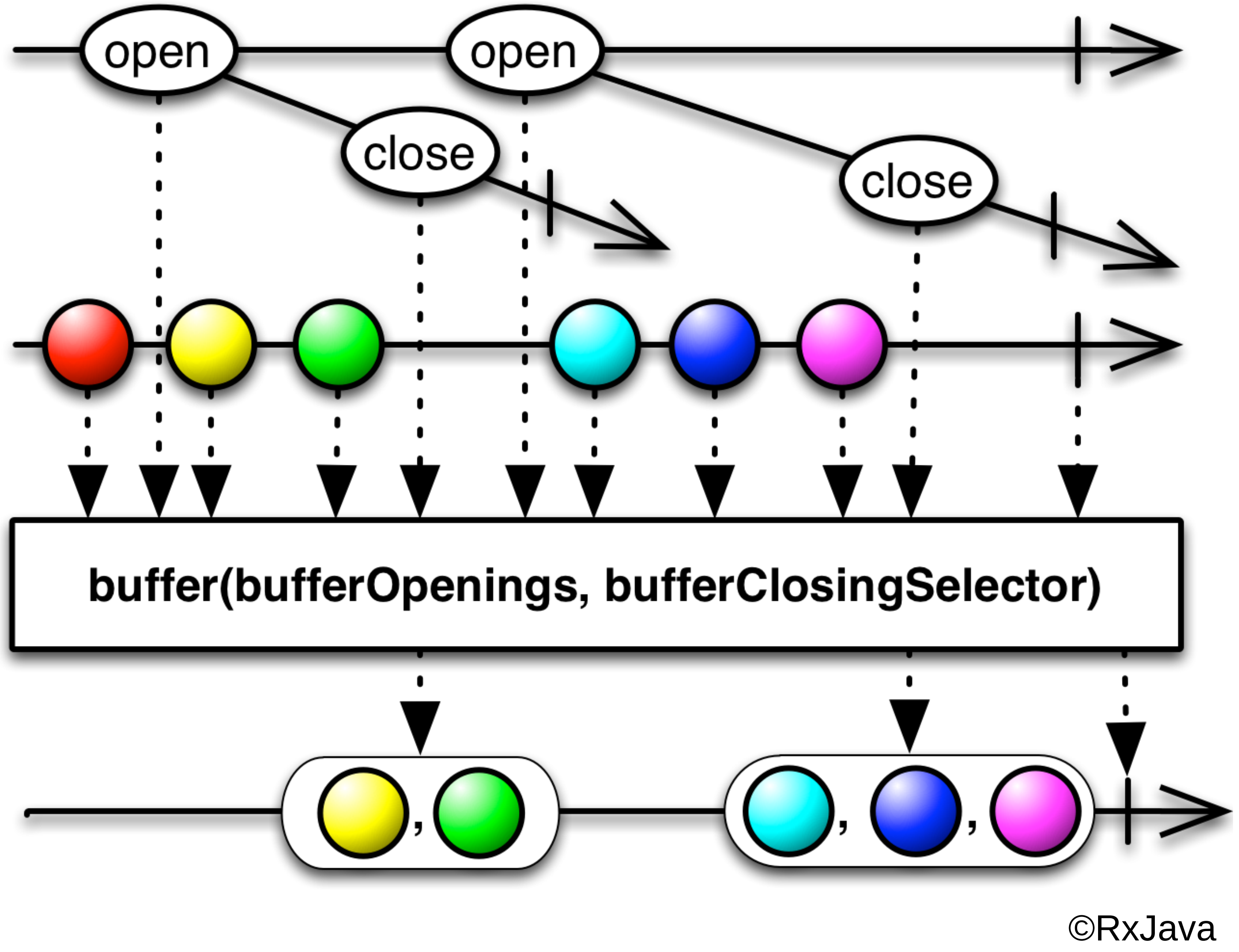
Windowing acts as a buffer, but emits Observables instead of lists of buffered items

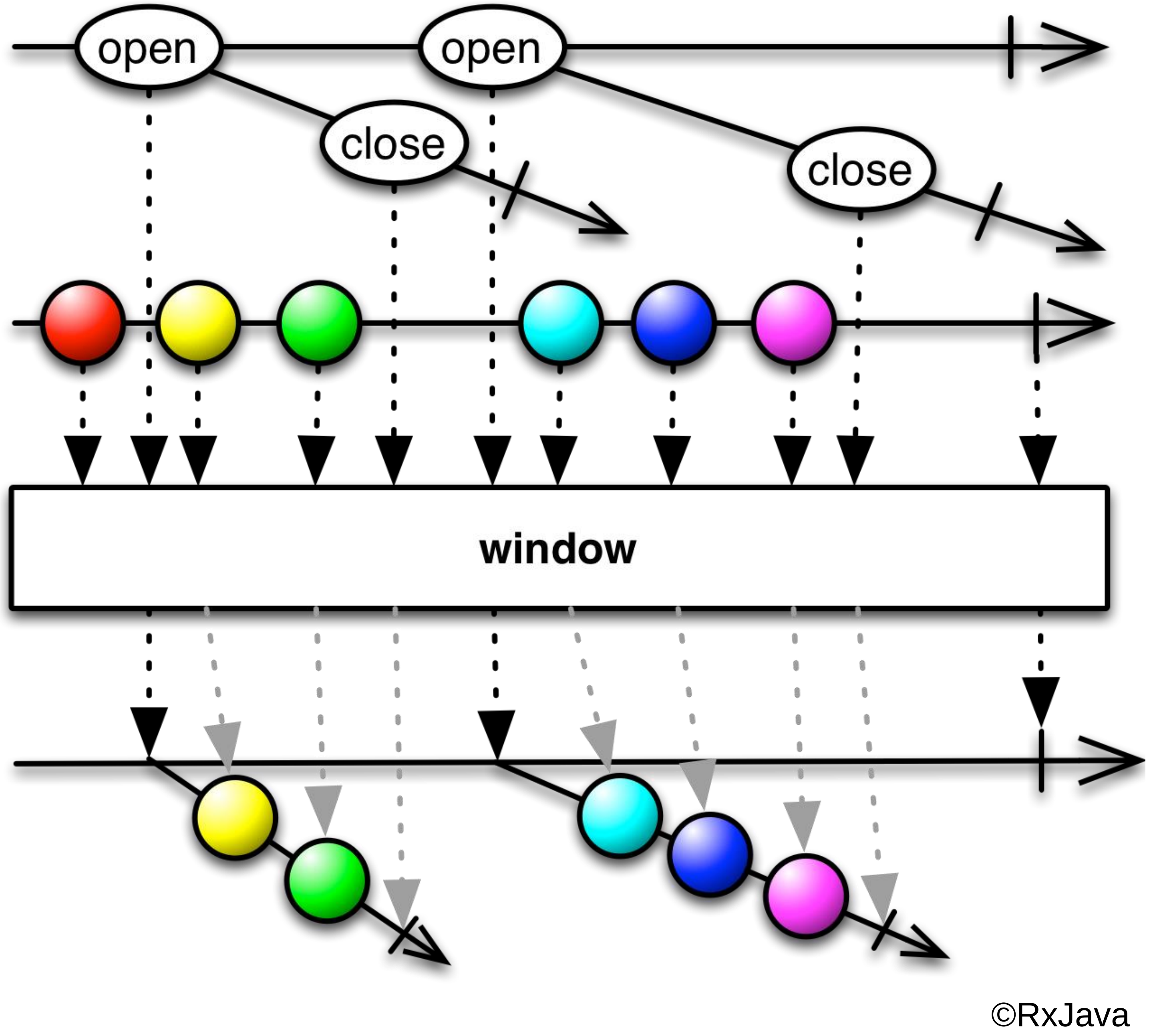
```
window(int size) { }                               // Observable<Observable<T>>

window(long timeSpan, TimeUnit unit) { } // Observable<Observable<T>>
window(long timeSpan, TimeUnit unit, int maxSize) { }

window(Observable<O> bufferOpenings,                         // Openings events
       Func1<O, Observable<C>> bufferClosings) { } // Closings events
```

Buffering & windowing

Windowing acts as a buffer, but emits Observables instead of lists of buffered items

Buffering & windowing





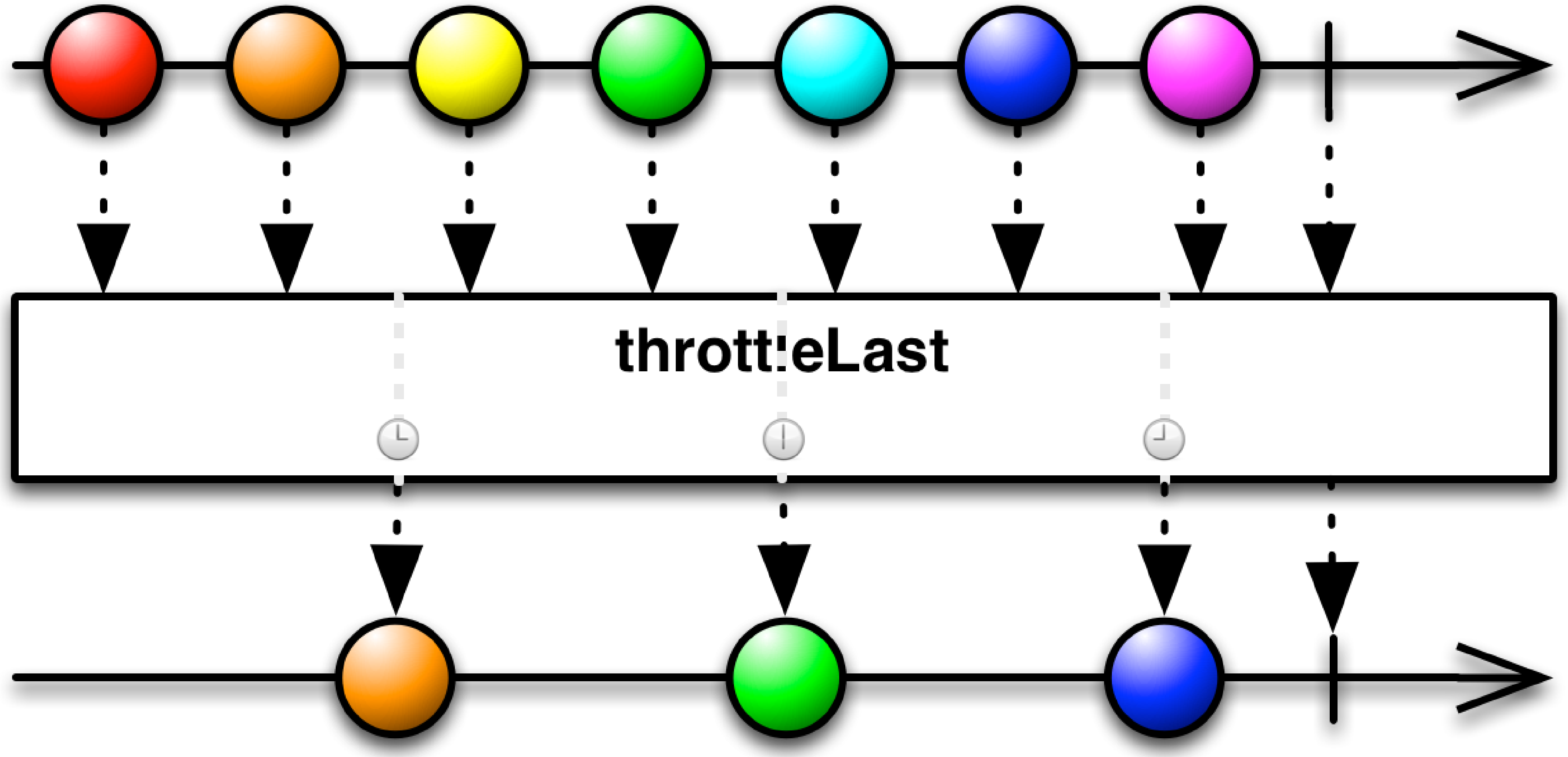
Throttling

Throttling is a sampler on the beginning or the end of a window

```
throttleFirst(long windowDuration, TimeUnit unit) { } // Observable<T>

throttleLast(long windowDuration, TimeUnit unit) { }

throttleWithTimeout(long windowDuration, TimeUnit unit) { }
```

Throttling

Throttling is a sampler on the beginning or the end of a window



©RxJava

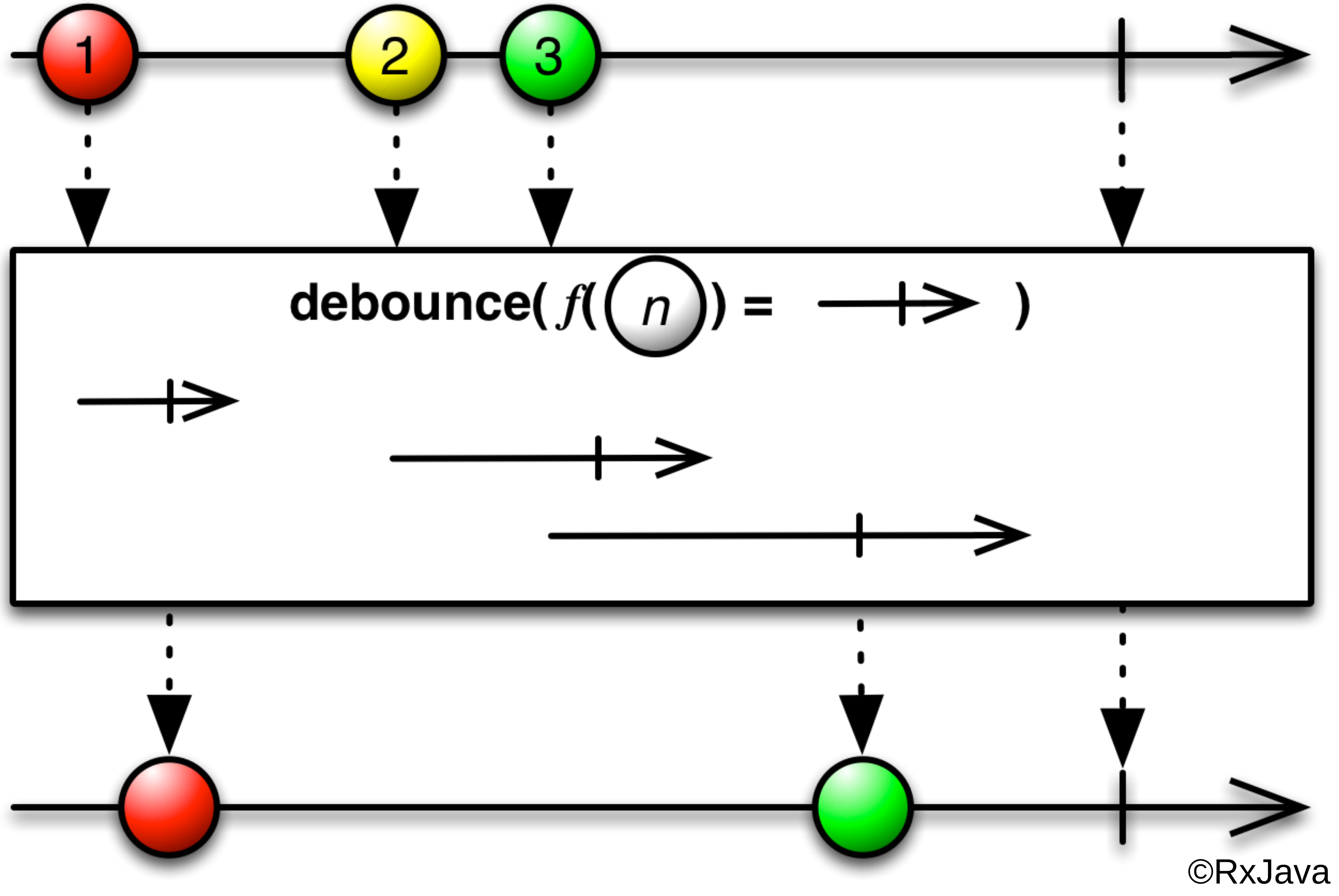
Debouncing

Debounce also limits the rate of the emission of the items, by adding a delay before reemitting

```
debounce(long delay, TimeUnit timeUnit) ; // Observable<T>

debounce(Func1<T, Observable<U> func1) ;  // Observable<T>
```

Debouncing

It two items are emitted within the same time frame, the first one is lost



Wrap-up on RxJava

Complex API, many different concepts, many methods

Allows to process data in chosen threads, this is useful for IO, computations, specialized threads (GUI threads)

Allows the synchronization of operations

- on clocks
- on application events

Works in pull mode, and also in push mode

- backpressure

Getting the best of both worlds?

Getting the best of both API?

What about connecting a J8 Stream to a Rx Observable?

Getting the best of both API?

If we have an Iterator, this is easy:

```
Iterator<T> iterator = ... ;

Observable<T> observable = Observable.from(() -> iterator) ;
```

Getting the best of both API?

If we have an Iterator, this is easy:

```
Iterator<T> iterator = ... ;

Observable<T> observable = Observable.from(() -> iterator) ;
```

If we have a Spliterator, not much more complex:

```
Spliterator<T> spliterator = ... ;

Observable<T> observable =
   Observable.from(() -> Spliterators.iterator(spliterator)) ;
```

Getting the best of both API?

So if we have a Stream, we can easily build an Observable

Getting the best of both API?

So if we have a Stream, we can easily build an Observable

What about the other way?

Getting the best of both API?

So if we have a Stream, we can easily build an Observable

What about the other way?

1) We can build an Iterator on an Observable
2) Then build a Spliterator on an Iterator

Getting the best of both API?

So if we have a Stream, we can easily build an Observable

What about the other way?

1) We can build an Iterator on an Observable
2) Then build a Spliterator on an Iterator

But we need to do that ourselves…

Iterating on an Observable

To implement an Iterator:

1) We need to implement next() and hasNext()
2) remove() is a default method in Java 8

Iterating on an Observable

The trick is that

- an iterator pulls the date from a source
- an observable pushes the data to callbacks

Iterating on an Observable

The trick is that

- an iterator pulls the date from a source
- an observable pushes the data to callbacks

So we need an adapter…

Iterating on an Observable

How has it been done in the JDK?

```
public static<T> Iterator<T>
iterator(Spliterator<? extends T> spliterator) {

    class Adapter implements Iterator<T>, Consumer<T> {
        // implementation
    }

    return new Adapter() ;
}
```

```java
class Adapter implements Iterator<T>, Consumer<T> {
   boolean valueReady = false ;
   T nextElement;

   public void accept(T t) {
      valueReady = true ;
      nextElement = t ;
   }

   public boolean hasNext() {
      if (!valueReady)
         spliterator.tryAdvance(this) ; // calls accept()
      return valueReady ;
   }

   public T next() {
      if (!valueReady && !hasNext())
         throw new NoSuchElementException() ;
      else {
         valueReady = false ;
         return nextElement ;
      }
   }
}
```

Iterating on an Observable

Let us adapt this pattern!

```
public static<T> Iterator<T>
of(Observable<? extends T> observable) {

    class Adapter implements Iterator<T> {
        // implementation
    }

    return new Adapter() ;
}
```

```java
class Adapter implements Iterator<T>, Consumer<T> {
   boolean valueReady = false ;
   T nextElement;

   public void accept(T t) { // needs to be called by the Observable
      valueReady = true ;
      nextElement = t ;
   }

   public boolean hasNext() {
      return valueReady ;
   }

   public T next() {
      if (!valueReady && !hasNext())
         throw new NoSuchElementException() ;
      else {
         valueReady = false ;
         return nextElement ;
      }
   }
}
```

Iterating on an Observable

The accept method

```
class Adapter implements Iterator<T>, Consumer<T> {
   boolean valueReady = false ;
   T nextElement;

   public void accept(T t) {
      observable.subscribe(
         element   -> nextElement = element, // onNext
         exception -> valueReady = false,    // onError
         () -> valueReady = false            // onComplete
      ) ;
   }
}
```

Iterating on an Observable

The accept method

final…

```
class Adapter implements Iterator<T>, Consumer<T> {
    boolean valueReady = false ;
    T nextElement;

    public void accept(T t) {
        observable.subscribe(
            element   -> nextElement = element, // onNext
            exception -> valueReady = false,    // onError
            () -> valueReady = false            // onComplete
        ) ;
    }
}
```

Iterating on an Observable

We can wrap those value in Atomic variable

```
class Adapter implements Iterator<T>, Consumer<T> {
   AtomicBoolean valueReady = new AtomicBoolean(false) ;
   AtomicReference<T> nextElement = new AtomicReference() ;

   public void accept(T t) {
      observable.subscribe(
         element   -> nextElement.set(element), // onNext
         exception -> valueReady.set(false),    // onError
         () -> valueReady.set(false)            // onComplete
      ) ;
   }
}
```

Iterating on an Observable

Cant we do better?

```
interface Wrapper<E> {

   E get() ;




}
```

```
Wrapper<Boolean> wb = () -> true ;
```

Iterating on an Observable

Cant we do better?

```
interface Wrapper<E> {

   E get() ;




}
```

```
Wrapper<Boolean> wb = () -> true ;
Action1<Boolean> onNext = b -> wb.set(b) ; // should return Wrapper<T>
```

Iterating on an Observable

Cant we do better?

```
interface Wrapper<E> {

    E get() ;

    public default Wrapper<E> set(E e) {
        // should return a wrapper of e
    }
}
```

```
Wrapper<Boolean> wb = () -> true ;
Action1<Boolean> onNext = b -> wb.set(b) ; // should return Wrapper<T>
```

Iterating on an Observable

Cant we do better?

```
interface Wrapper<E> {

   E get() ;

   public default Wrapper<E> set(E e) {
      return () -> e ;
   }
}
```

```
Wrapper<Boolean> wb = () -> true ;
Action1<Boolean> onNext = b -> wb.set(b) ; // should return Wrapper<T>
```

Iterating on an Observable

We can wrap those value in Atomic variable

```
class Adapter implements Iterator<T>, Consumer<T> {
   Wrapper<Boolean> valueReady = () -> false ;
   Wrapper<T> nextElement ;

   public void accept(T t) {
      observable.subscribe(
         element   -> nextElement.set(element), // onNext
         exception -> valueReady.set(false),    // onError
         () -> valueReady.set(false)            // onComplete
      ) ;
   }
}
```

Iterating on an Observable

So we can build an Iterator on an Observable

And with it, a Spliterator on an Observable

it will work on *cold* observables

Getting the best of both API

The cold observables can be implemeted with Java 8 Streams

The hot observables can be implemented by combining both API

Comparisons: Patterns & Performances

Let us do some comparisons

Let us take one use case

Implemented in Rx and J8 Streams

See the different ways of writing the same processings
And compare the processing times using JMH

Shakespeare plays Scrabble

Published in Java Magazine

Presented in Java One 2014

Used during Virtual Technology Summit

https://community.oracle.com/docs/DOC-916777

https://github.com/JosePaumard/jdk8-lambda-tour

Shakespeare plays Scrabble

Basically, we have the set of the words used by Shakespeare

The official Scrabble player dictionnary

And the question is: how good at Scrabble Shakespeare would have been?

Shakespeare plays Scrabble

1) Build the histogram of the letters of a word
2) Number of blanks needed for a letter in a word
3) Number of blanks needed to write a word
4) Predicate to check is a word can be written with 2 blanks
5) Bonus for a doubled letter
6) Final score of a word
7) Histogram of the scores
8) Best word

1) Histogram of the letters

Java 8 Stream API – Rx Java

```
// Histogram of the letters in a given word
Function<String, Map<Integer, Long>> histoOfLetters =
      word -> word.chars().boxed()
                  .collect(
                     Collectors.groupingBy(
                        Function.identity(),
                        Collectors.counting()
                     )
                  ) ;
```

1) Histogram of the letters

Java 8 Stream API – Rx Java

```
// Histogram of the letters in a given word
Func1<String, Observable<HashMap<Integer, LongWrapper>>> histoOfLetters =
        word -> toIntegerObservable.call(word)
                 .collect(
                    () -> new HashMap<Integer, LongWrapper>(),
                    (HashMap<Integer, LongWrapper> map, Integer value) -> {
                       LongWrapper newValue = map.get(value) ;
                       if (newValue == null) {
                          newValue = () -> 0L ;
                       }
                       map.put(value, newValue.incAndSet()) ;
                    }) ;
```

1) Histogram of the letters

Java 8 Stream API – Rx Java

```
interface LongWrapper {
    long get() ;

    public default LongWrapper set(long l) {
        return () -> l ;
    }

    public default LongWrapper incAndSet() {
        return () -> get() + 1L ;
    }

    public default LongWrapper add(LongWrapper other) {
        return () -> get() + other.get() ;
    }
}
```

2) # of blanks for a letter

Java 8 Stream API – Rx Java

```
// number of blanks for a given letter
ToLongFunction<Map.Entry<Integer, Long>> blank =
    entry ->
        Long.max(
            0L,
            entry.getValue() -
                scrabbleAvailableLetters[entry.getKey() - 'a']
        ) ;
```

2) # of blanks for a letter

Java 8 Stream API – Rx Java

```
// number of blanks for a given letter
Func1<Entry<Integer, LongWrapper>, Observable<Long>> blank =
    entry ->
        Observable.just(
            Long.max(
                0L,
                entry.getValue().get() -
                    scrabbleAvailableLetters[entry.getKey() - 'a']
```

3) # of blanks for a word

Java 8 Stream API – Rx Java

```
// number of blanks for a given word
Function<String, Long> nBlanks =
    word -> histoOfLetters.apply(word)
                          .entrySet().stream()
                          .mapToLong(blank)
                          .sum();
```

3) # of blanks for a word

Java 8 Stream API – Rx Java

```
// number of blanks for a given word
Func1<String, Observable<Long>> nBlanks =
   word -> histoOfLetters.call(word)
                 .flatMap(map -> Observable.from(() ->
                                        map.entrySet().iterator()))
                 .flatMap(blank)
                 .reduce(Long::sum) ;
```

4) Predicate for 2 blanks

Java 8 Stream API – Rx Java

```
// can a word be written with 2 blanks?
Predicate<String> checkBlanks = word -> nBlanks.apply(word) <= 2 ;
```

```
// can a word be written with 2 blanks?
Func1<String, Observable<Boolean>> checkBlanks =
   word -> nBlanks.call(word)
                  .flatMap(l -> Observable.just(l <= 2L)) ;
```

5) Bonus for a doubled letter – 1

Java 8 Stream API – Rx Java

```
// Placing the word on the board
// Building the streams of first and last letters
Function<String, IntStream> first3 =
   word -> word.chars().limit(3);
```

5) Bonus for a doubled letter – 1

Java 8 Stream API – Rx Java

```
// Placing the word on the board
// Building the streams of first and last letters
Func1<String, Observable<Integer>> first3 =
   word ->
      Observable.from(
         IterableSpliterator.of(
            word.chars().boxed().limit(3).spliterator()
         )
      ) ;
```

5) Bonus for a doubled letter – 2

Java 8 Stream API – Rx Java

```
// Bonus for double letter
ToIntFunction<String> bonusForDoubleLetter =
    word -> Stream.of(first3.apply(word), last3.apply(word))
                  .flatMapToInt(Function.identity())
                  .map(scoreOfALetter)
                  .max()
                  .orElse(0) ;
```

5) Bonus for a doubled letter – 2

Java 8 Stream API – Rx Java

```
// Bonus for double letter
Func1<String, Observable<Integer>> bonusForDoubleLetter =
   word -> Observable.just(first3.call(word), last3.call(word))
                     .flatMap(observable -> observable)
                     .flatMap(scoreOfALetter)
                     .reduce(Integer::max) ;
```

6) Final score of a word

Java 8 Stream API – Rx Java

```
// score of the word put on the board
Function<String, Integer> score3 =
   word ->
      2*(score2.apply(word)
      + bonusForDoubleLetter.applyAsInt(word))
      + (word.length() == 7 ? 50 : 0);
```

6) Final score of a word

Java 8 Stream API – Rx Java

```
// score of the word put on the board
Func1<String, Observable<Integer>> score3 =
   word ->
      Observable.just(
         score2.call(word), score2.call(word),
         bonusForDoubleLetter.call(word),
         bonusForDoubleLetter.call(word),
         Observable.just(word.length() == 7 ? 50 : 0)
      )
      .flatMap(observable -> observable)
      .reduce(Integer::sum) ;
```

7) Histogram of the scores

Java 8 Stream API – Rx Java

```
Function<Function<String, Integer>, Map<Integer, List<String>>>
buildHistoOnScore =
    score -> shakespeareWords.stream().parallel()
                        .filter(scrabbleWords::contains)
                        .filter(checkBlanks)
                        .collect(
                            Collectors.groupingBy(
                                score,
                                () -> new TreeMap<>(Comparator.reverseOrder()),
                                Collectors.toList()
                            )
                         ) ;
```

7) Histogram of the scores

Java 8 Stream API – Rx Java

```
Func1<Func1<String, Observable<Integer>>, Observable<TreeMap<Integer, List<String>>>>
buildHistoOnScore =
        score -> Observable.from(() -> shakespeareWords.iterator())
                           .filter(scrabbleWords::contains)
                           .filter(word -> checkBlanks.call(word).toBlocking().first())
                           .collect(
                                () -> new TreeMap<Integer, List<String>>(Comparator.reverseOrder()),
                                (TreeMap<Integer, List<String>> map, String word) -> {
                                     Integer key = score.call(word).toBlocking().first() ;
                                     List<String> list = map.get(key) ;
                                     if (list == null) {
                                         list = new ArrayList<String>() ;
                                         map.put(key, list) ;
                                     }
                                     list.add(word) ;
                           }) ;
```

8) Best word

Java 8 Stream API – Rx Java

```
// best key / value pairs
List<Entry<Integer, List<String>>> finalList =
   buildHistoOnScore.apply(score3).entrySet()
                    .stream()
                    .limit(3)
                    .collect(Collectors.toList()) ;
```

8) Best word

Java 8 Stream API – Rx Java

```
// best key / value pairs
List<Entry<Integer, List<String>>> finalList2 =
   buildHistoOnScore.call(score3)
      .flatMap(map -> Observable.from(() -> map.entrySet().iterator()))
      .take(3)
      .collect(
         () -> new ArrayList<Entry<Integer, List<String>>>(),
         (list, entry) -> { list.add(entry) ; }
      )
      .toBlocking()
      .first() ;
```

8) Best word

Java 8 Stream API – Rx Java

```
// best key / value pairs
CountDownLatch latch = new CountDownLatch(3) ;
   buildHistoOnScore.call(score3)
      .flatMap(map -> Observable.from(() -> map.entrySet().iterator()))
      .take(3)
      .collect(
         () -> new ArrayList<Entry<Integer, List<String>>>(),
         (list, entry) -> { list.add(entry) ; latch.countDown() ; }
      )
     .forEach(...) ;

latch.await() ;
```

Patterns comparison

Java 8 Stream API: clean, simple, factory methods for Collectors

RxJava: flatMap calls, lack of factory methods

Java 8 can easily go parallel, which is a plus

Performances

Let us use JMH

Standard tool for measuring code performance on the JVM

Developed as an Open JDK tool

By Aleksey Shipilev http://shipilev.net/

https://twitter.com/shipilev

http://openjdk.java.net/projects/code-tools/jmh/

http://openjdk.java.net/projects/code-tools/jcstress/

https://www.parleys.com/tutorial/java-microbenchmark-harness-the-lesser-two-evils

Performances

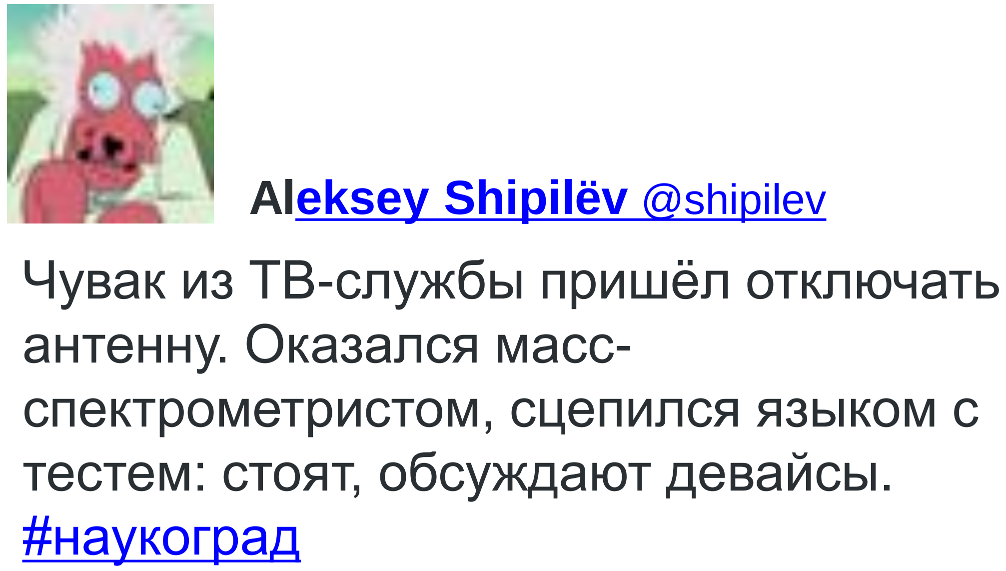
Let us use JMH

Standard tool for measuring code performance on the JVM

Developed as an Open JDK tool

By Aleksey Shipilev http://shipilev.net/

https://twitter.com/shipilev



http://openjdk.java.net/projects/code-tools/jmh/

http://openjdk.java.net/projects/code-tools/jcstress/

https://www.parleys.com/tutorial/java-microbenchmark-harness-the-lesser-two-evils

JMH

Easy to setup

```
<dependency>
   <groupId>org.openjdk.jmh</groupId>
   <artifactId>jmh-core</artifactId>
   <version>1.11.1</version>
</dependency>
```

JMH

Easy to use

```
@Benchmark
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@Warmup(iterations=5)
@Measurement(iterations=5)
@Fork(3)
public List<Entry<Integer, List<String>>> measureAverage() {

   // implementation to test
}
```

JMH

Launching a benchmark

```
> mvn clean install
> java –jar target/benchmark.jar
```

JMH

3 ways of measuring performances

- **average execution time**
- number of executions per second
- quantiles diagrams

Performances

Average execution time

```
Benchmark            Mode  Cnt     Score    Error  Units
NonParallelStreams   avgt  100   29,027 ± 0,279  ms/op
```

Performances

Average execution time

Benchmark	Mode	Cnt	Score		Error	Units
NonParallelStreams	avgt	100	29,027	±	0,279	ms/op
RxJava	avgt	100	253,788	±	1,421	ms/op

Performances

Average execution time

Benchmark	Mode	Cnt	Score		Error	Units
NonParallelStreams	avgt	100	29,027	±	0,279	ms/op
RxJava	avgt	100	253,788	±	1,421	ms/op
ParallelStreams	avgt	100	7,624	±	0,055	ms/op

Performances

Average execution time

Benchmark	Mode	Cnt	Score		Error	Units
NonParallelStreams	avgt	100	29,027	±	0,279	ms/op
RxJava	avgt	100	253,788	±	1,421	ms/op
ParallelStreams	avgt	100	7,624	±	0,055	ms/op

RxJava spends a lot of time openning observables, due to the all flatMap patterns

Conclusion

Conclusion

Functional programming is in the mood

From the pure performance point of view, things are not that simple

Java 8 Streams have adopted a partial functional approach, which is probably a good trade off

Conclusion

RxJava: rich and complex API

- many patterns are available in Java 8 Streams
- can run in Java 7 applications
- the « push » approach is very interesting
- choose your use case carefully, to avoir performance hits

http://www.reactive-streams.org/
http://reactivex.io/
https://github.com/reactive-streams/

Conclusion

Java 8 Streams: part of the JDK

- good performances, efficient memory footprint
- parallelization
- extensible through the Spliterator patterns

Conclusion

With more to come in Java 9

Java 9 is bringing a reactive framework as part of java.util.concurrent

http://openjdk.java.net/jeps/266
http://gee.cs.oswego.edu/dl/jsr166/dist/docs/index.html (Class Flow)


JavaOne™
Thank you


JavaOne™
Q / A