
JavaOne™
Going reactive
in Java 8 & 9
@JosePaumard

Agenda

1) To present the concepts on which the Stream API has been built
2) See the main patterns, what can be done with it

Agenda

1) To present the concepts on which the Stream API has been built
2) See the main patterns, what can be done with it

3) What is missing to become reactive?
4) What is in the work for Java 9?

Questions?

#J8RXS

José PAUMARD

MCF Um. Paris 13

PhD Appl C.S.

Open source dev.

Independant

@JosePaumard

José PAUMARD

pluralsight
hardcore dev and IT training

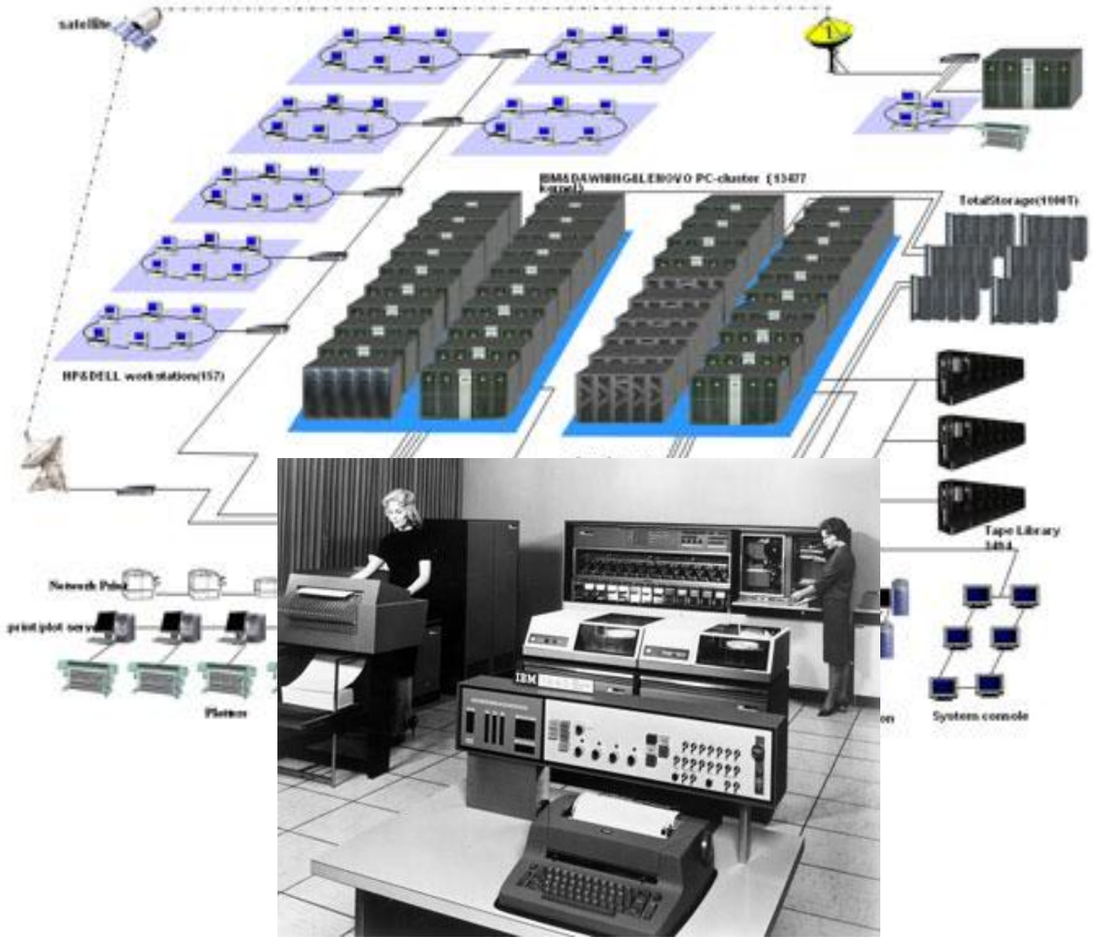
Java™ magazine
By and for the Java community

Parleys

Microsoft Virtual Academy

@JosePaumard

Data Processing



Java 8
Stream API

What is a Stream?

- A new concept in Java 8
- An interface (or several interfaces)

- Goals:
  - To provide an implemenation of the map / filter / reduce
  - Simple to use
  - Efficient (memory, computation)

Definition of a Stream

Two things about streams:

1) A Stream does not hold any data
2) A Stream does not modify the data it gets from the source

Patterns to create a Stream

- There are many patterns to create a Stream

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

Patterns to create a Stream

- There are many patterns to create a Stream

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

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

Patterns to create a Stream

- There are many 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);
```

Patterns to create a Stream

- There are many 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);

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

Patterns to create a Stream

- There are many 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);

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

Stream<String> lines = Files.lines(Paths.get("alice-in-wonderland.txt"));
```

Patterns to use a Stream

- Map filter reduce with the Stream API

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

double average = people.stream()
    .filter(person -> person.getCity().equals("San Francisco"))
    .mapToInt(Person::getAge)
    .filter(age -> age > 20)
    .average().get();
```

- « a Stream does not hold any data »

Patterns to use a Stream

- Map filter reduce with the Stream API

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

double average = people.stream()     // Stream<Person>
    .filter(person -> person.getCity().equals("San Francisco"))
    .mapToInt(Person::getAge)
    .filter(age -> age > 20)
    .average().get();
```

- « a Stream does not hold any data »

Patterns to use a Stream

- Map filter reduce with the Stream API

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

double average = people.stream()     // Stream<Person>
   .filter(person -> person.getCity().equals("San Francisco"))
   .mapToInt(Person::getAge)         // IntStream
   .filter(age -> age > 20)
   .average().get();
```

- « a Stream does not hold any data »

Patterns to use a Stream

- Map filter reduce with the Stream API

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

double average = people.stream()      // Stream<Person>
    .filter(person -> person.getCity().equals("San Francisco"))
    .mapToInt(Person::getAge)         // IntStream
    .filter(age -> age > 20)          // IntStream
    .average().get();
```

- « a Stream does not hold any data »

Patterns to use a Stream

- Map filter reduce with the Stream API

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

double average = people.stream()      // Stream<Person>
    .filter(person -> person.getCity().equals("San Francisco"))
    .mapToInt(Person::getAge)         // IntStream
    .filter(age -> age > 20)          // IntStream
    .average().get();                 // double
```

- « a Stream does not hold any data »

Patterns to use a Stream

- Map filter reduce with the Stream API

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

double average = people.stream()      // Stream<Person>
    .filter(person -> person.getCity().equals("San Francisco"))
    .mapToInt(Person::getAge)         // IntStream
    .filter(age -> age > 20)          // IntStream
    .average().get();                 // double
```

- An operation that returns a Stream does not process any data

« a Stream does not hold any data »

« a Stream does not hold any data » is a very powerful paradygm

- It brings the notion of *lazyness*
- And optimizations!

Back to our initial program

- Efficient (memory, computation)

- Two things about streams:

1) A Stream does not hold any data
2) A Stream does not modify its data

Back to our initial program

- **Efficient (memory, computation)**

- Two things about streams:

1) **A Stream does not hold any data**
2) A Stream does not modify its data

Back to our initial program

- Efficient (memory, computation)

- Two things about streams:

1) A Stream does not hold any data
2) **A Stream does not modify its data**

Back to our initial program

- Efficient (memory, computation)

- Two things about streams:

1) A Stream does not hold any data
2) **A Stream does not modify its data → Parallelism!**

Going parallel

- Back to our previous example

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

double average = people.stream().parallel()
    .filter(person -> person.getCity().equals("San Francisco"))
    .mapToInt(Person::getAge)
    .filter(age -> age > 20)
    .average().get();
```

What about non-standard sources?

A Stream is built on two things:

- A Spliterator (split – iterator)

What about non-standard sources?

A Stream is built on two things:

- A Spliterator (split – iterator)
- A ReferencePipeline (the implementation)

The Spliterator

- The Spliterator holds a special word: `characteristics`

The Spliterator

- The Spliterator holds a special word: `characteristics`

```
public interface Spliterator<T> {

   public static final int ORDERED     = 0x00000010;
   public static final int DISTINCT    = 0x00000001;
   public static final int SORTED      = 0x00000004;
   public static final int SIZED       = 0x00000040;
   public static final int NONNULL     = 0x00000100;
   public static final int IMMUTABLE   = 0x00000400;
   public static final int CONCURRENT  = 0x00001000;
   public static final int SUBSIZED    = 0x00004000;
}
```

The Spliterator

- The Spliterator holds a special word: `characteristics`

```
// ArrayListSpliterator
public int characteristics() {
    return Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED;
}
```

```
// HashMap.KeySpliterator
public int characteristics() {
    return (fence < 0 || est == map.size ? Spliterator.SIZED : 0) |
                  Spliterator.DISTINCT;
}
```

The Spliterator

- The Spliterator holds a special word: `characteristics`

- This word is used for optimization

```
people.stream()
    .sorted() // quicksort?
    .collect(Collectors.toList());
```

The Spliterator

- The Spliterator holds a special word: `characteristics`

- This word is used for optimization

```
people.stream()
    .sorted() // quicksort? It depends on SORTED == 0
    .collect(Collectors.toList());
```

The Spliterator

- The Spliterator holds a special word: `characteristics`

- This word is used for optimization

```
SortedSet<Person> people = ...;

people.stream()
    .sorted() // SORTED == 1, no quicksort
    .collect(Collectors.toList());
```

The Spliterator

- The Spliterator holds a special word: `characteristics`

- This word is used for optimization

```
ArrayList<Person> people = ...;

people.stream()
    .sorted() // SORTED == 0, quicksort
    .collect(Collectors.toList());
```

The characteristics can change

- Each Stream object in a pipeline has its own characteristics

The characteristics can change

- Each Stream object in a pipeline has its own characteristics

Method	Set to 0	Set to 1
filter()	SIZED	-
map()	DISTINCT, SORTED	-
flatMap()	DISTINCT, SORTED, SIZED	-
sorted()	-	SORTED, ORDERED
distinct()	-	DISTINCT
limit()	SIZED	-
peek()	-	-
unordered()	ORDERED	-

The characteristics can change

- Each Stream object in a pipeline has its own characteristics

Method	Set to 0	Set to 1
filter()	**SIZED**	-
map()	DISTINCT, SORTED	-
flatMap()	DISTINCT, SORTED, SIZED	-
sorted()	-	SORTED, ORDERED
distinct()	-	DISTINCT
limit()	SIZED	-
peek()	-	-
unordered()	ORDERED	-

The characteristics can change

- Each Stream object in a pipeline has its own characteristics

Method	Set to 0	Set to 1
filter()	SIZED	-
map()	**DISTINCT, SORTED**	-
flatMap()	DISTINCT, SORTED, SIZED	-
sorted()	-	SORTED, ORDERED
distinct()	-	DISTINCT
limit()	SIZED	-
peek()	-	-
unordered()	ORDERED	-

What about non-standard sources?

- The Spliterator is meant to be overriden

```
public interface Spliterator<T> {

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

   Spliterator<T> trySplit() ;

   long estimateSize();

   int characteristics();
}
```

What about non-standard sources?

- 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
}
```

Spliterators on spliterators

- Building a Spliterator on another Spliterator allows:

Grouping: `[1, 2, 3, 4, 5, …]` ->

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

Spliterators on spliterators

- Building a Spliterator on another Spliterator allows:

Rolling: `[1, 2, 3, 4, 5, …]` ->

`[[1, 2, 3], [2, 3, 4], [3, 4, 5], …]`

Spliterators on spliterators

- Building a Spliterator on another Spliterator allows:

Zipping: `[1, 2, 3, …], [a, b, c, …]` ->

`[F[1, a], F[2, b], F[3, c], …]`

Spliterators on spliterators

- Building a Spliterator on another Spliterator allows:

Zipping `[1, 2, 3, …], [a, b, c, …]` ->
+ grouping: `[[F[1, a], F[2, b], F[3, c]],`
`[F[4, d], F[5, e], F[6, f]], …]`

Spliterators on spliterators

- Building a Spliterator on another Spliterator allows:

Zipping
+ rolling:

```
[1, 2, 3, …], [a, b, c, …]  ->
  [[F[1, a], F[2, b], F[3, c]],
   [F[2, b], F[3, c], F[4, d]], …]
```

Java 8 Stream API

- Simple, readable patterns
- Fast and efficient (with more to come)
- A Stream looks like a Collection, but it is not

- The Spliterator can be implemented to connect a Stream to « non-standard » sources of data
- Or to change the way the data is analyzed

Java 8
Stream API
and beyond

Java 8 Stream API

Back to the definitions:

1) A Stream does not hold any data
2) A Stream does not modify its data

Java 8 Stream API

Back to the definitions:

1) A Stream does not hold any data
2) A Stream does not modify its data

How does a Stream work?

1) It connects to a source of data: one source = one stream
2) It consumes the data from the source: « pull mode »

Java 8 Stream API

What about:

- Connecting several streams to a single source?

Java 8 Stream API

What about:

- Connecting several streams to a single source?
- Connecting several sources to a single stream?

Java 8 Stream API

What about:

- Connecting several streams to a single source?
- Connecting several sources to a single stream?

- Having a source that produces data whether or not a stream is connected to it

Java 8 Stream API

What about:

- Connecting several streams to a single source?
- Connecting several sources to a single stream?

- Having a source that produces data whether or not a stream is connected to it

Clearly, the Stream API has not been made to handle this

Reactive Stream API

- This leads to the « reactive stream » API
- 3rd party API: Rx Java (and several other languages)
- Implementations available as a preview of JDK 9
  Everything takes place in `java.util.concurrent.Flow`
  Available on the JSR166 web site

Push mode stream

- Let us write a model for the source of data

```
public interface Publisher<T> {

   public ...  subscribe(Subscriber<T> subscriber);
}
```

Push mode stream

- Let us write a model for the source of data

```
public interface Publisher<T> {

   public ...  subscribe(Subscriber<T> subscriber);
}
```

- As a subscriber I will want to unsubscribe
- So I need an object from the publisher
  on which I can call `cancel()`

Push mode stream

- Let us write a model for the source of data

```
public interface Publisher<T> {

   public Subscription subscribe(Subscriber<T> subscriber);
}
```

- The first idea that could come to mind is to return a `Subscription` object

Push mode stream

- Let us write a model for the source of data

```
public interface Publisher<T> {

   public void subscribe(Subscriber<T> subscriber);
}
```

- But it will be a callback, to stay in an asynchronous world

Push mode stream

- Callback in the subscriber to get a subscription

```
public interface Subscriber<T> {

   public void onSubscribe(Subscription subscription);
}
```

Push mode stream

- Callback in the subscriber to get a subscription

```
public interface Subscriber<T> {

   public void onSubscribe(Subscription subscription);
}
```

```
public interface Subscription {

   public void cancel();
}
```

Push mode stream

- The publisher might look like this

```
public class SimplePublisher<T> implements Publisher<T> {

   private Set<Subscriber<T>> subscribers = ConcurrentHashMap.newKeySet();

   public void subscribe(Subscriber<T> subscriber) {

      if (subscribers.add(subscriber)) {
         Subscription subscription = new SimpleSubscription();
         subscriber.onSubscribe(subscription);
      }
   }
}
```

Push mode stream

- In the subscribing code

```
public class SimpleSubscriber<T> implements Subscriber<T> {

   private Subscription subscription;

   @Override
   public void onSubscribe(Subscription subscription) {
      this.subscription = subscription;
   }
}
```

Push mode stream

- In the running code

```
Publisher<String> publisher = ...;
Subscriber<String> subscriber = ...;

publisher.subscribe(subscriber);

// some more code

subscriber.getSubscription().cancel();
```

Push mode stream

- Callback in the subscriber to get a subscription

```
public interface Subscriber<T> {

   public void onSubscribe(Subscription subscription);
}
```

- I also need callbacks to get the data itself

Push mode stream

- Callback in the subscriber to get a subscription

```
public interface Subscriber<T> {

   public void onSubscribe(Subscription subscription);









}
```

Push mode stream

- Callback in the subscriber to get a subscription

```
public interface Subscriber<T> {

   public void onSubscribe(Subscription subscription);

   public void onNext(T item);




}
```

Push mode stream

- Callback in the subscriber to get a subscription

```
public interface Subscriber<T> {

   public void onSubscribe(Subscription subscription);

   public void onNext(T item);

   public void onComplete();


}
```

Push mode stream

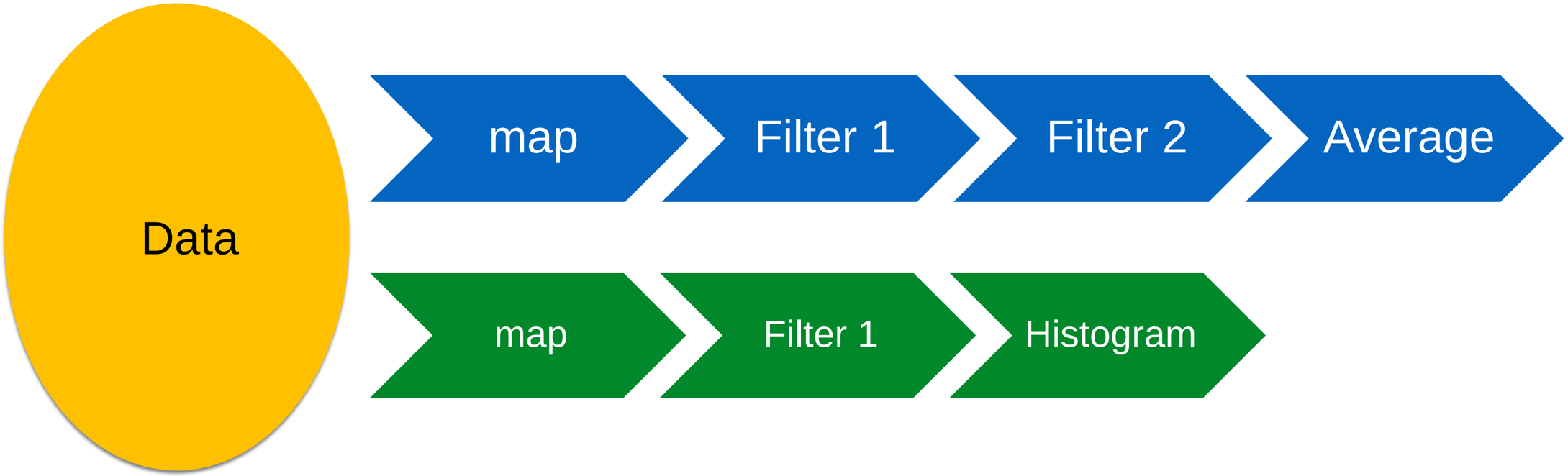
- Callback in the subscriber to get a subscription

```
public interface Subscriber<T> {

   public void onSubscribe(Subscription subscription);

   public void onNext(T item);

   public void onComplete();

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

Push mode stream

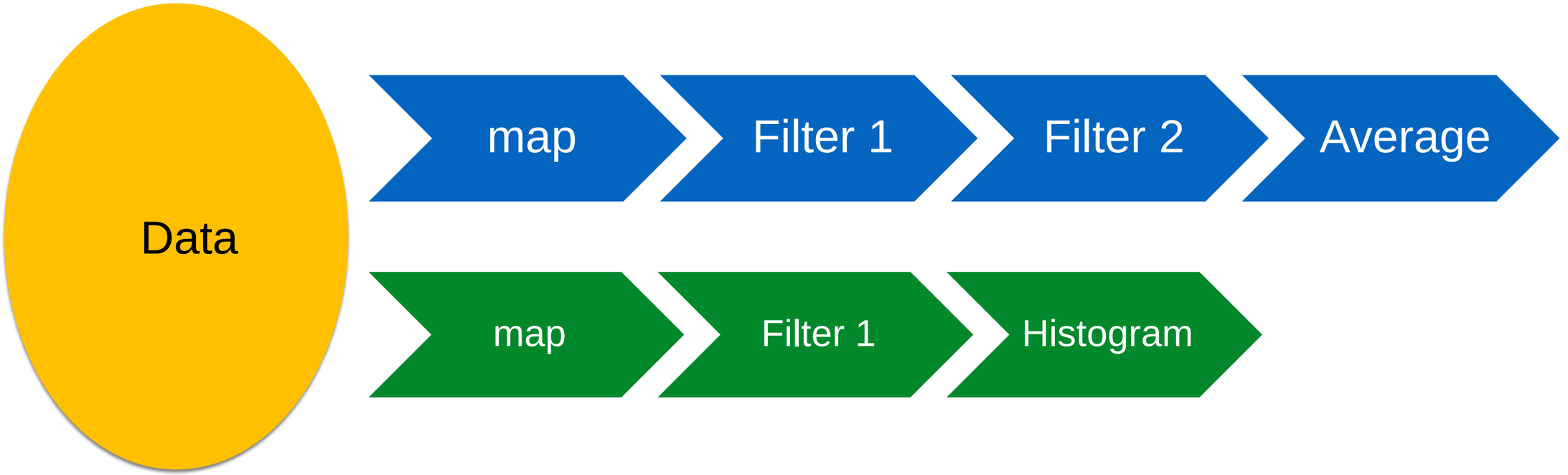
- Having a source that produces data independantly from its consumers implies to work in an asynchronous mode
- The API is built on callbacks

Several streams per source



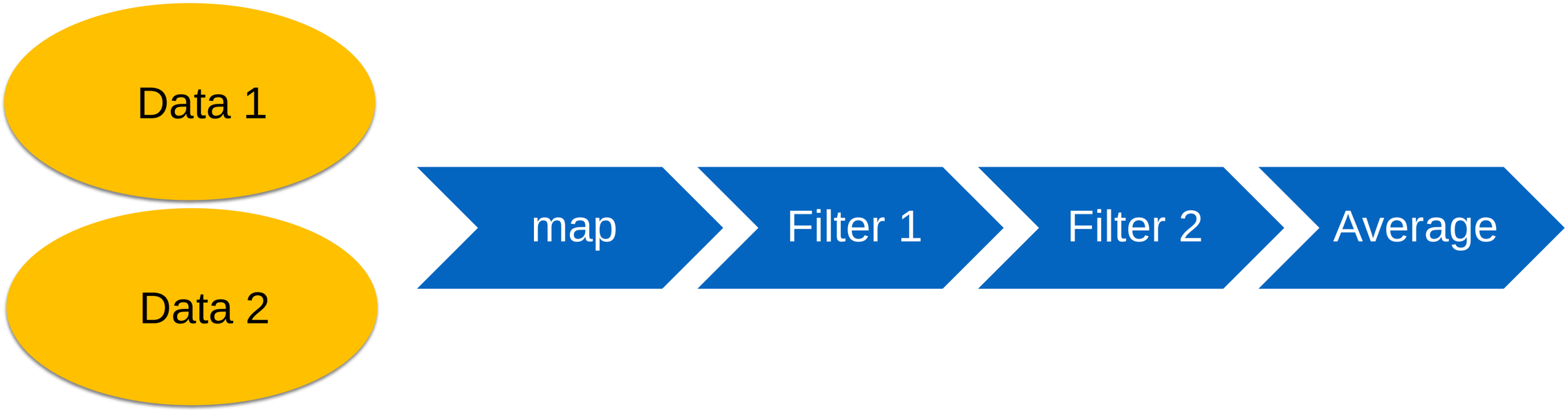
- In « pull mode », it would not work, or would require the streams to be synchronized

Several streams per source



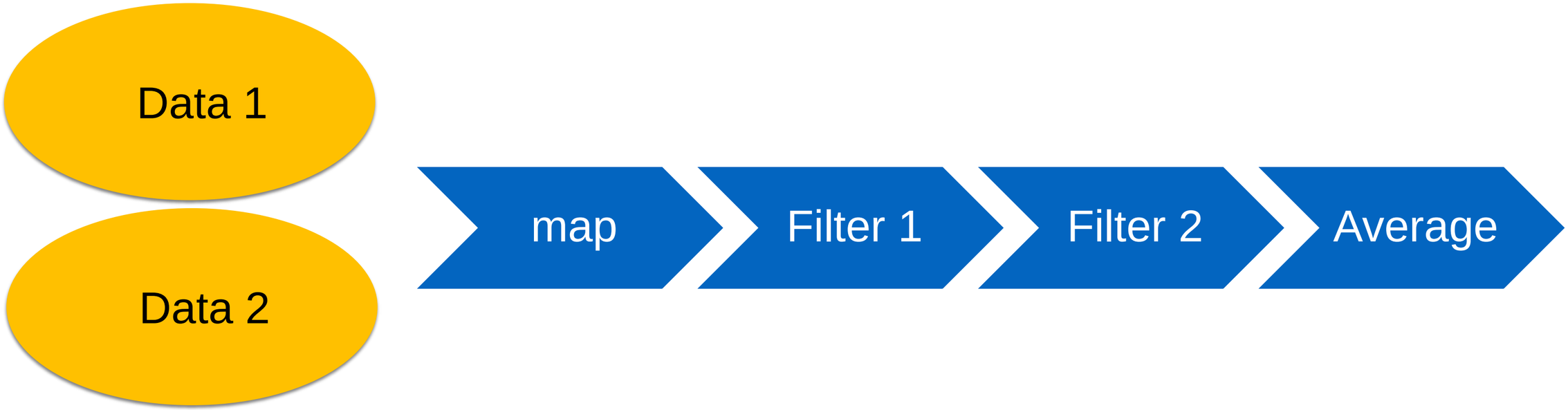
- In « pull mode », it would not work, or would require the streams to be synchronized
- In « push mode », it does not raise any problem

Several sources for a stream



- In « pull mode », it requires a special Spliterator

Several sources for a stream



- In « pull mode », it requires a special Spliterator
- In « push mode », since both sources are not synchronized, we may have problems

Push mode with several sources

- At some point in our data processing pipeline we want to see both sources as one, *ie* merged in some way

Push mode with several sources

- At some point in our data processing pipeline we want to see both sources as one, *ie* merged in some way

- How can we merge them if one source is faster than the other?

Push mode with several sources

- At some point in our data processing pipeline we want to see both sources as one, *ie* merged in some way
- How can we merge them if one source is faster than the other?
- Several strategies are possible

Merging sources in push mode

1) Decide to follow one of the data publishers, the first one

Merging sources in push mode

1) Decide to follow one of the data publishers, the first one

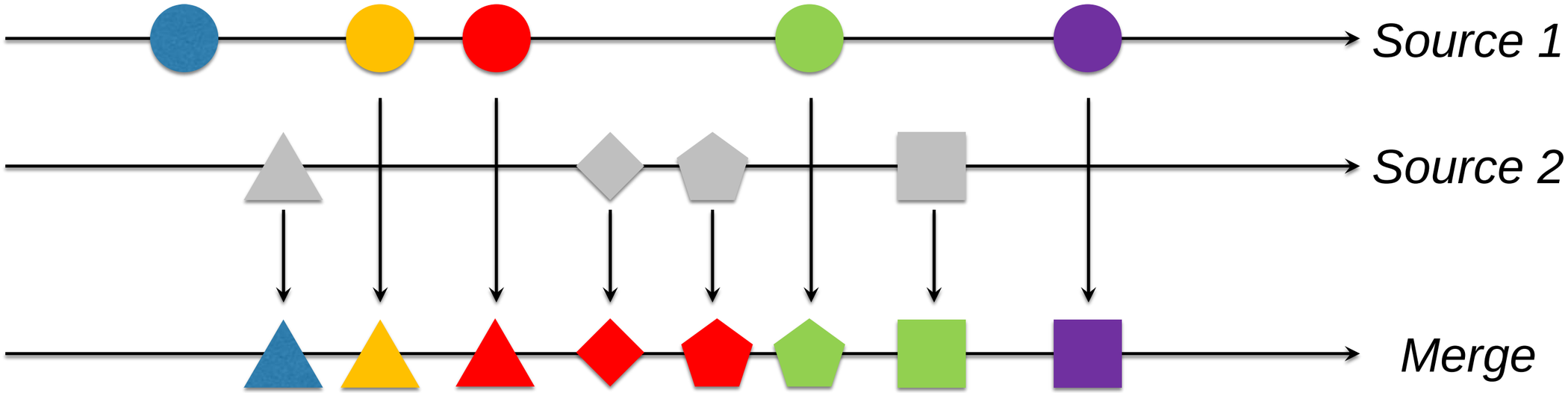
   Use case: identical requests on several DNS, or on several Rest Services
   The first to give the answer is the winner!
   And makes the others useless

Merging sources in push mode

1) Decide to follow one of the streams, the first one
2) Combine the two last seen items,
   everytime a new item is generated

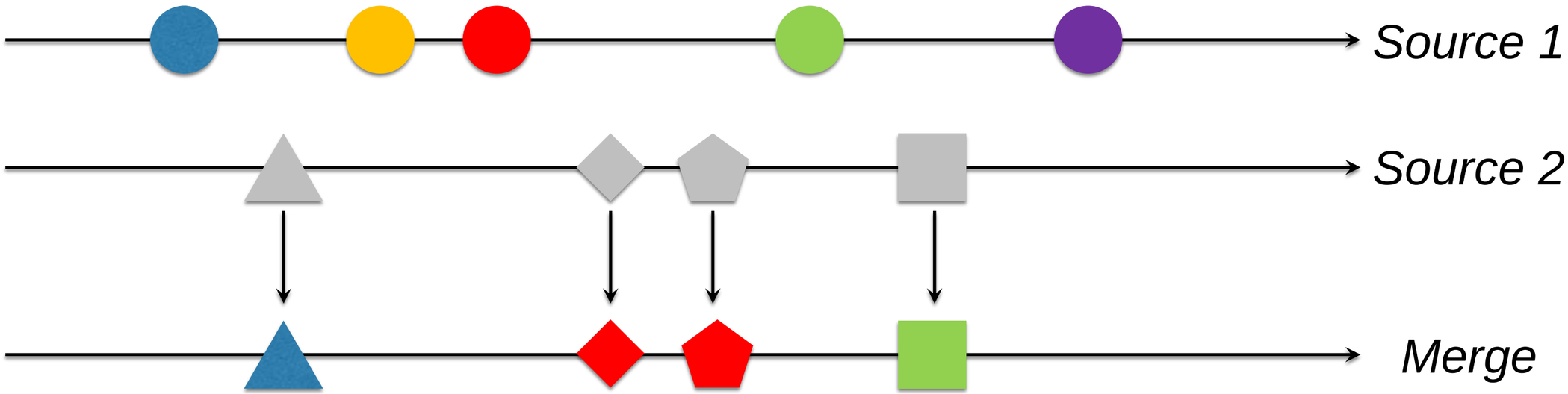
Merging sources in push mode

1) Decide to follow one of the streams, the first one
2) Combine the two last seen items,
   everytime a new item is generated



Merging sources in push mode

1) Decide to follow one of the streams, the first one
2) Combine the two last seen items,
   or synchronized on the second source (for instance)

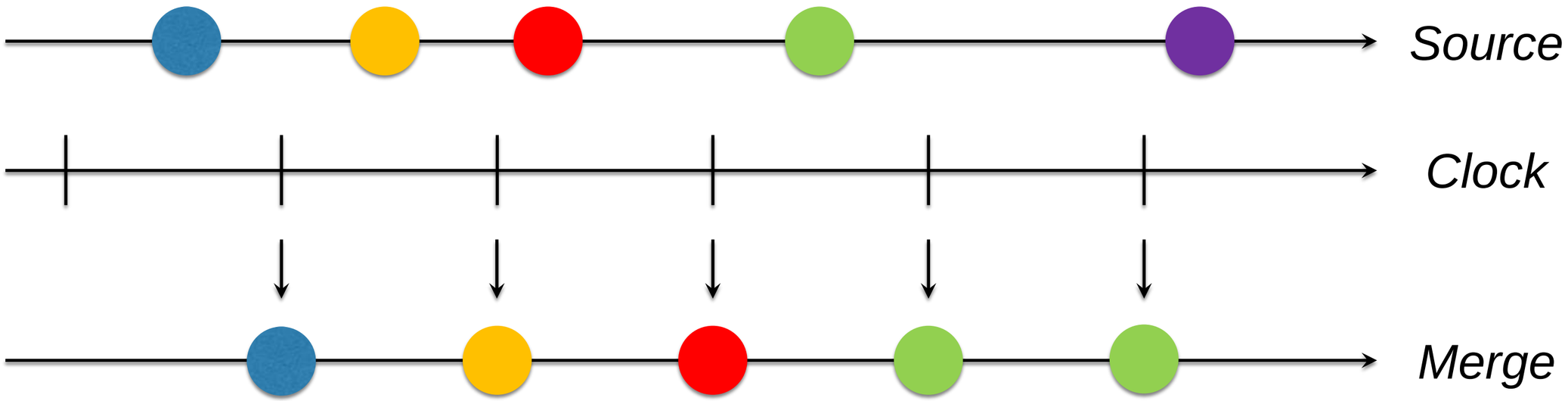


Merging sources in push mode

- This second approach brings the idea of synchronizing on a source
- A source can play the role of a clock

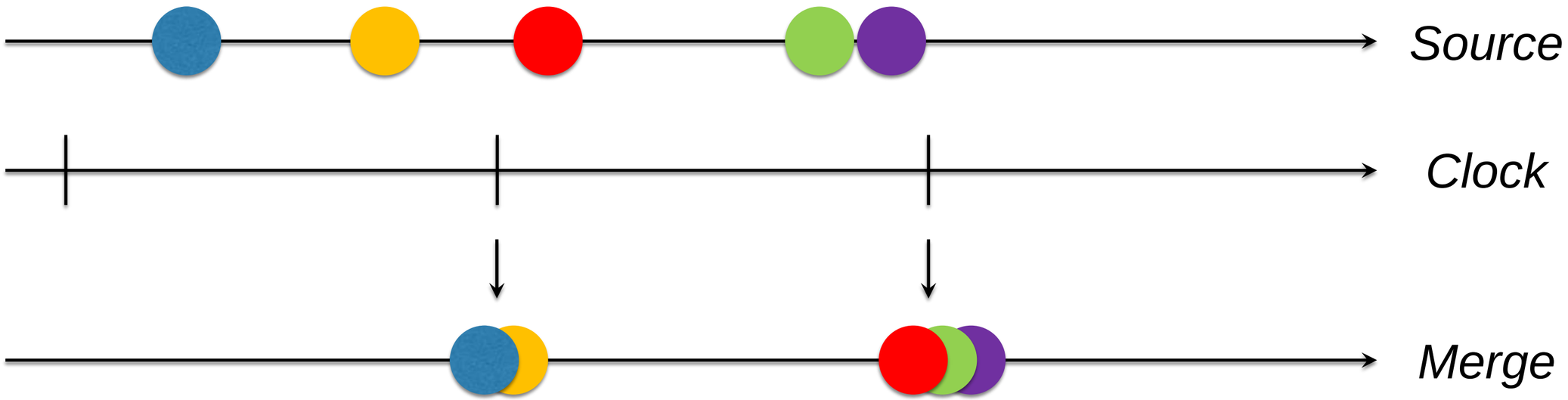
Merging sources in push mode

- Let us build a sampler



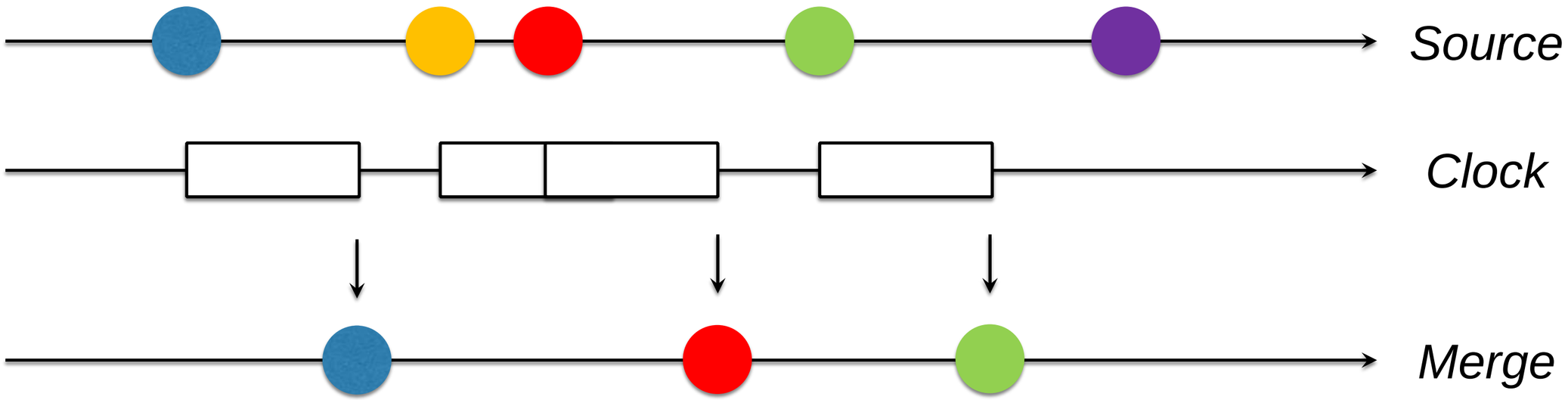
Merging sources in push mode

- Let us build a sampler with a function



Merging sources in push mode

- Or a « debouncer »



Merging sources in push mode

- There is no limit to what can be done with two independant sources merged into one

- The synchronization-on-a-clock can be used to « slow down » a source

A central question

- What will happen if a source is « too fast »?
- That is, a consumer cannot process data fast enough
- It leads to the question of « backpressure »

Backpressure

- Several strategies:

1) Create a buffer

Backpressure

- Several strategies:

1) Create a buffer
2) Synchronize on a clock, or a gate, that could be generated by the slow observer and sample, or windows, or debounce, or…

Backpressure

- Several strategies:

1) Create a buffer
2) Synchronize on a clock, or a gate, that could be generated by the slow observer and sample, or windows, or debounce, or…
3) Try to slow down the source (can be done if I have the hand on both the producer and the consumer)

Backpressure

- There is code for that in the `Subscription` object

```
public interface Subscription {

    public void cancel();

    public void request(long n);
}
```

- The `request()` method is there to give information to the producer

Backpressure

- There is code for that in the `Subscription` object

```
public void onNext(String element) {

   // process the element

   this.subscription.request(1L);
}
```

- The `request()` method is there to give information to the producer

Backpressure

- Several strategies:

1) Create a buffer
2) Synchronize on a clock, or a gate, that could be generated by the slow observer and sample, or windows, or debounce, or…
3) Try to slow down the source (can be done if I have the hand on both the producer and the consumer)
4) Have several observers in parallel and then merge the results

Reactive Streams

- New concept (at least in the JDK)
- New complexity, several use cases are possible

- Still under work (in the JDK and in 3rd party)

Reactive Streams links

- Some references on the reactive streams:

  - http://www.reactive-streams.org/

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

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

Reactive Streams links

- In the classes currently available in the JSR 166 package:
- The class `Flow` has the `Publisher`, `Subscriber` and `Subscription` interfaces, and the `Processor` interface
- The class `SubmissionPublisher`, that implements `Publisher`, meant to be overriden or used as a component of a complete implementation

Conclusion

- Java 8 Stream API: great API to process data in a « pull » mode
- The introduction of a « push » mode allows for many improvements (synchronization, backpressure)
- The backpressure question is relevant
- Loosing items ≠ loosing information!

Conclusion

- Streams & Reactive Streams are very active topics
- Java Stream has been released with Java 8, improvements will be added in Java 9 and beyond
- Reactive Streams has several 3rd party implementations (RxJava) in several languages
- Will be part of Java 9


JavaOne™
Thank you


JavaOne™
Q/A