Streams & Collectors

New APIs for map / filter / reduce

José Paumard
blog.paumard.org
@JosePaumard

pluralsight
hardcore dev and IT training

Module Outline

- Introduction: map / filter / reduce
- What is a « Stream »?
- Patterns to build a Stream
- Operations on a Stream

Map / Filter / Reduce

- Example:
- Let's take a list a Person

```
List<Person> list = new ArrayList<>() ;
```

- Suppose we want to compute the

« average of the age of the people older than 20 »

1. Convert Object :: Person --> Age
2. Select People older than 20 Years
3. Compute average age.

Map / Filter / Reduce

- 1st step: mapping
- The mapping step takes a List<Person> and returns a List<Integer>
- The size of both lists is the same

Map / Filter / Reduce

- 2nd step: filtering
- The filtering step takes a List<Integer> and returns a List<Integer>
- But there some elements have been filtered out in the process

Map / Filter / Reduce

- 3rd step: average
- This is the reduction step, equivalent to the SQL aggregation

What Is a Stream?

- Technical answer: a typed interface

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {

        // ...
}
```

- And a new concept!

What Is a Stream?

- **What does it do?**

- **It gives ways to efficiently process large amounts of data… and also smaller ones**

Using Java Stream, we can process large and small amounts of data.

What Is a Stream?

- What does *efficiently* mean?

Parallel + Pipeline

- Two things:
- In parallel, to leverage the computing power of multicore CPUs
- Pipelined, to avoid unnecessary intermediary computations

What Is a Stream?

- So what is a Stream?
- An object on which one can define *operations*
- An object that does not hold any data
- An object that should not change the data it processes
- An object able to process data in « one pass »
- An object optimized from the algorithm point of view, and able to process data in parallel

Why Stream does not hold any data?

Beacause an operation on a stream that returns another stream is called an intermediary operation which is a simple declaration.

Note: any Terminal operation like forEach() / reduction operation actually process data.

How Can We Build a Stream?

- Many patterns!

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

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

A First Operation

ForEach() that takes a Consumer <T>

- First operation: forEach()

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

Stream<Person> stream = persons.stream();
stream.forEach(p -> System.out.println(p));
```

- Prints all the elements of the list
- It takes an instance of Consumer as an argument

void accept (T t) = (p -> System.out.println(p));

Consumer c = new Consumer(){
void accept(Person p){System.out.println(p.name())}
}

p -> System.out.println(p.getName());

A First Operation

- Interface Consumer<T>

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);
}
```

- Consumer<T> is a *functional interface*
- Can be implemented by a lambda expression

```
Consumer<T> c = p -> System.out.println(p);
```

```
Consumer<T> c = System.out::println; // Method reference
```

A First Operation

- In fact Consumer<T> is a bit more complex

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

- One can chain consumers!

A First Operation

- Let's chain consumers

```
List<String> list = new ArrayList<>();

Consumer<String> c1 = s -> list.add(s);
Consumer<String> c2 = s -> System.out.println(s);
```

A First Operation

- Let's chain consumers

```
List<String> list = new ArrayList<>();

Consumer<String> c1 = list::add;
Consumer<String> c2 = System.out::println;

Consumer<String> c3 = c1.andThen(c2);
```

Consumer Chaining

A First Operation

- Only way to have several consumers on a single stream

```
List<String> result = new ArrayList<>();
List<Person> persons = ...;

Consumer<String> c1 = result::add;
Consumer<String> c2 = System.out::println;

persons.stream()
       .forEach(c1.andThen(c2));
```

- Because forEach() does not return anything

A Second Operation: Filter

Filter that takes a Predicate

- Example:

```
List<Person> list = ...;
Stream<Person> stream = list.stream();
Stream<Person> filtered =
    stream.filter(person -> person.getAge() > 20);
```

- Takes a predicate as a parameter:

```
Predicate<Person> p = person -> person.getAge() > 20;
```

A Second Operation: Filter

- **Predicate interface:**

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);
}
```

```
Predicate p = new Predicate <Person>() {

    boolean test (Person person) {

        return person.getAge() > 30 ;

    }
}
```

```
person -> person.getAge() > 30;
```

```
stream.filter( p)
```

A Second Operation: Filter

- Predicate interface, with default methods:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);

  default Predicate<T> and(Predicate<? super T> other) { ... }

  default Predicate<T> or(Predicate<? super T> other) { ... }

  default Predicate<T> negate() { ... }
}
```

Filter Logic

A Second Operation: Filter

- Predicates combinations examples:

```
Predicate<Integer> p1 = i -> i > 20;
Predicate<Integer> p2 = i -> i < 30;
Predicate<Integer> p3 = i -> i == 0;

Predicate<Integer> p = p1.and(p2).or(p3); // (p1 AND p2) OR p3
Predicate<Integer> p = p3.or(p1).and(p2); // (p3 OR p1) AND p2
```

- Warning: method calls do not handle priorities

A Second Operation: Filter

- Predicate interface, with static method:

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    // default methods

    static <T> Predicate<T> isEqual(Object o) { ... }
}
```

A Second Operation: Filter

- Predicate interface, with static method:

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    // default methods

    static <T> Predicate<T> isEqual(Object o) { ... }
}
```

- Example:

```
Predicate<String> p = Predicate.isEqual("two") ;
```

A Second Operation: Filter

- Use case:

```
Predicate<String> p = Predicate.isEqual("two") ;

Stream<String> stream1 = Stream.of("one", "two", "three") ;

Stream<String> stream2 = stream1.filter(p) ;
```

- The filter method returns a Stream

A Second Operation: Filter

- Use case:

```
Predicate<String> p = Predicate.isEqual("two") ;

Stream<String> stream1 = Stream.of("one", "two", "three") ;

Stream<String> stream2 = stream1.filter(p) ;
```

- The filter method returns a Stream --> stream2
- This Stream is a new instance stream1 =/ stream2

both are different instances

A Second Operation: Filter

- Question: what do I have in this new Stream?
- Simple answer: the filtered data

- Really?
- We just said: « a stream does not hold any data »

A Second Operation: Filter

- Question: what do I have in this new Stream?
- Simple answer: ~~the filtered data~~ WRONG!
- The right answer is: nothing, since a Stream does not hold any data
- So, what does this code do?

```
List<Person> list = ... ;
Stream<Person> stream = list.stream();
Stream<Person> filtered =
        stream.filter(person -> person.getAge() > 20);
```

- Answer is: nothing

This call is only a declaration, no data is processed

A Second Operation: Filter

- The call to the filter method is *lazy*
- And all the methods of Stream that return another Stream are *lazy*
- Another way of saying it:

an operation on a Stream that returns a Stream is called an intermediary operation

Back to the Consumer

- What does this code do?

```
List<String> result = new ArrayList<>();
List<Person> persons = ... ;

persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(result::add);
```

Back to the Consumer

- What does this code do?

```
List<String> result = new ArrayList<>();
List<Person> persons = ... ;

persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(result::add);
```

- Hint: the peek() method returns a Stream

Back to the Consumer

- What does this code do?

```
List<String> result = new ArrayList<>();
List<Person> persons = ... ;

persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(result::add);
```

- Answer: nothing!
- This code does not print anything
- The list « result » is empty

Summary

- The Stream API defines *intermediary operations*
- We saw 3 operations:
  - forEach(Consumer) (not lazy)
  - peek(Consumer) (lazy)
  - filter(Predicate) (lazy)

forEach() is not a lazy operation

Mapping Operation

- Example:

```
List<Person> list = ... ;
Stream<Person> stream = list.stream();
Stream<String> names =
        stream.map(person -> person.getName());
```

Mapping Operation

- Example:

```
List<Person> list = ... ;
Stream<Person> stream = list.stream();
Stream<String> names =
        stream.map(person -> person.getName());
```

- map() returns a Stream, so it is an intermediary operation

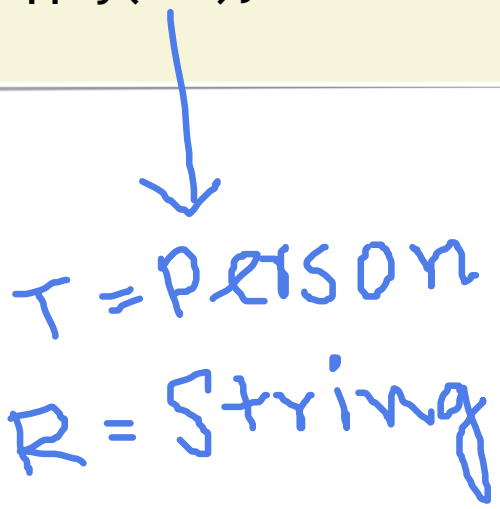
map() basically alters the type of data that a stream contains.

Mapping Operation

- A mapper is modeled by the Function interface

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);
}
```



Mapping Operation

- … with default methods to chain and compose mappings

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    default <V> Function<V, R> compose(Function<V, T> before);

    default <V> Function<T, V> andThen(Function<R, V> after);
}
```

Mapping Operation

- … with default methods to chain and compose mappings

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    default <V> Function<V, R> compose(Function<V, T> before);

    default <V> Function<T, V> andThen(Function<R, V> after);
}
```

- In fact this is the simplified version, beware the generics!

We can chain map operations.

Mapping Operation

- compose() and andThen() methods with their exact signatures

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    default <V> Function<V, R> compose(
        Function<? super V, ? extends T> before);

    default <V> Function<T, V> andThen(
        Function<? super R, ? extends V> after);
}
```

Mapping Operation

- One static method: identity

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    // default methods

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

Flatmapping Operation

- Method flatMap()
- Signature:

```
<R> Stream<R> flatMap(Function<T, Stream<R>> flatMapper);
```

```
<R> Stream<R> map(Function<T, R> mapper);
```

Flatmapping Operation

- Method flatMap()
- Signature:

```
<R> Stream<R> flatMap(Function<T, Stream<R>> flatMapper);
```

```
<R> Stream<R> map(Function<T, R> mapper);
```

- The flatMapper takes an element of type T, and returns an element of type Stream<R>

Flatmapping Operation

- Method flatMap()
- Signature:

```
<R> Stream<R> flatMap(Function<T, Stream<R>> flatMapper);
```

```
 <R> Stream<R> map(Function<T, R> mapper);
```

- If the flatMap was a regular map, it would return a Stream<Stream<R>>

Flatmapping Operation

- Method flatMap()
- Signature:

```
<R> Stream<R> flatMap(Function<T, Stream<R>> flatMapper);
```

```
 <R> Stream<R> map(Function<T, R> mapper);
```

- If the flatMap was a regular map, it would return a Stream<Stream<R>>
- Thus a « stream of streams »

Flatmapping Operation

- Method flatMap()
- Signature:

```
<R> Stream<R> flatMap(Function<T, Stream<R>> flatMapper);
```

```
 <R> Stream<R> map(Function<T, R> mapper);
```

- If the flatMap was a regular map, it would return a Stream<Stream<R>>
- But it is a flatMap!

Flatmapping Operation

- Method flatMap()
- Signature:

```
<R> Stream<R> flatMap(Function<T, Stream<R>> flatMapper);
```

```
<R> Stream<R> map(Function<T, R> mapper);
```

- If the flatMap was a regular map, it would return a Stream<Stream<R>>
- But it is a flatMap!
- Thus the « stream of streams » is flattened, and becomes a stream

Summary

- 3 categories of operations:
  - forEach() and peek()
  - filter()
  - map() and flatMap()

Reduction

- And what about the reduction step?
- Two kinds of reduction in the Stream API
- 1st: aggregation = min, max, sum, etc…

Reduction

- How does it work?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream();
Integer sum =
        stream.reduce(0, (age1, age2) -> age1 + age2);
```

- 1st argument: identity element of the reduction operation
  = Initial Value
- 2nd argument: reduction operation, of type BinaryOperator<T>

For any reduction, you need to pass the identity along with a biFunction

BinaryOperator

- A BinaryOperator is a special case of BiFunction

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);

    // plus default methods
}
```

```
@FunctionalInterface
public interface BinaryOperator<T>
extends BiFunction<T, T, T> {

    // T apply(T t1, T t2);

    // plus static methods
}
```

Identity Element

- The bifunction takes two arguments, so…
- What happens if the Stream is empty?
- What happens if the Stream has only one element?

- The reduction of an empty Stream is the identity element = Initial Value
- If the Stream has only one element, then the reduction is that element

Aggregations

- **Examples:**

```
Stream<Integer> stream = ...;
BinaryOperation<Integer> sum = (i1, i2) -> i1 + i2;
Integer id = 0; // identity element for the sum

int red = stream.reduce(id, sum);
```

identity + biFunction

```
Stream<Integer> stream = Stream.empty();

int red = stream.reduce(id, sum);
System.out.println(red);
```

Since the stream is empty, the value of i1 + i2 = 0

- **Will print:**

```
> 0
```

Stream empty() creates an empty sequential Stream.

Aggregations

- **Examples:**

```
Stream<Integer> stream = ...;
BinaryOperation<Integer> sum = (i1, i2) -> i1 + i2;
Integer id = 0; // identity element for the sum

int red = stream.reduce(id, sum);
```

```
Stream<Integer> stream = Stream.of(1);

int red = stream.reduce(id, sum);
System.out.println(red);
```

Returns a sequential ordered stream whose elements are the specified values

- **Will print:**

```
> 1
```

Stream <Integer> intStream = Stream.of(2);
Stream <Integer> intStream1 = Stream.of(2,4,6);

Aggregations

- Examples:

```
Stream<Integer> stream = ...;
BinaryOperation<Integer> sum = (i1, i2) -> i1 + i2;
Integer id = 0; // identity element for the sum

int red = stream.reduce(id, sum);
```

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4);

int red = stream.reduce(id, sum);
System.out.println(red);
```

- Will print:

```
> 10
```

Aggregations: Corner Case

- Suppose the reduction is the max

```
BinaryOperation<Integer> max =
    (i1, i2) ->
    i1 > i2 ? i1 : i2;
```

- The problem is, there is no identity element for the max reduction
- So the max of an empty Stream is undefined…

Aggregations: Corner Case

- Then what is the return type of this call?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream();
... max =
        stream.max(Comparator.naturalOrder());
```

Aggregations: Corner Case

- Then what is the return type of the this call?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream();
... max =
        stream.max(Comparator.naturalOrder());
```

- If it is an int, then the default value is 0…

Aggregations: Corner Case

- Then what is the return type of the this call?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream();
... max =
        stream.max(Comparator.naturalOrder());
```

- If it is an Integer, then the default value is null…

Optionals

- **Then what is the return type of the this call?**

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream();
Optional<Integer> max =
        stream.max(Comparator.naturalOrder());
```

- **Optional means « there might be no result »**

Another Example:

Optional <Integer> maxNumber= intStream1.max((i1, i2) -> i1 > i2 ? i1 :i2);

Optionals

- How to use an Optional?

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
        String s = opt.get() ;
} else {
        ...
}
```

- The method isPresent() returns true if there is something in the optional

Optionals

- How to use an Optional?

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
        String s = opt.get() ;
} else {
        ...
}
```

- The method isPresent() returns true if there is something in the optional
- The method get() returns the value held by this optional

Optionals

- How to use an Optional?

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
        String s = opt.get() ;
} else {
        ...
}
```

opt.orElse(opt.get()))

- The method orElse() encapsulates both calls

```
String s = opt.orElse("") ; // defines a default value
```

Optional <Integer> maxNumber= intStream1.max((i1, i2) -> i1 > i2 ? i1 :i2);

maxNumber.orElse(maxNumber.get());

Optionals

- How to use an Optional?

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
        String s = opt.get() ;
} else {
        ...
}
```

- The method orElseThrow() defines a thrown exception

```
String s = opt.orElseThrow(MyException::new) ; // lazy construct.
```

Reductions

- **Available reductions:**
  - max(), min()
  - count()
- **Boolean reductions**
  - allMatch(), noneMatch(), anyMatch()
- **Reductions that return an optional**
  - findFirst(), findAny()

Reductions

- Reductions are *terminal operations*
- They trigger the processing of the data

Terminal Operation

- Example:

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

Optional<Integer> minAge =
persons.map(person -> person.getAge())   // Stream<Integer>
       .filter(age -> age > 20)          // Stream<Integer>
       .min(Comparator.naturalOrder());  // terminal operation
```

Terminal Operation

- Example, optimization:

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

persons.map(person -> person.getLastName())
       .allMatch(length < 20);                // terminal op.
```

Terminal Operation

- Example, optimization:

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

persons.map(person -> person.getLastName())
       .allMatch(length < 20);                    // terminal op.
```

- The map / filter / reduce operations are evaluated in one pass over the data

Summary

- Reduction seen as an aggregation
- Intermediary / terminal operation
- Optional: needed because default values cant be always defined

Collectors

- There is another type of reduction
- Called « mutable » reduction
- Instead of aggregating elements, this reduction put them in a « container »

Collecting in a String

- Example:

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

String result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.joining(", ")
       );
```

- Result is a String with all the names of the people in persons, older than 20, separated by a comma

Collecting in a List

- Example:

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

List<String> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.toList()
       );
```

- Result is a List of String with all the names of the people in persons, older than 20

Collecting in a Map

- Example:

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

Map<Integer, List<Person>> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .collect(
          Collectors.groupingBy(Person::getAge)
       );
```

- Result is a Map containing the people of persons, older than 20
  - The keys are the ages of the people
  - The values are the lists of the people of that age

Collecting in a Map

- Example:

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

Map<Integer, List<Person>> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .collect(
          Collectors.groupingBy(Person::getAge)
       );
```

- It is possible to « post-process » the values, with a *downstream collector*

Collecting in a Map

- Example:

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

Map<Integer, Long> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .collect(
          Collectors.groupingBy(
             Person::getAge,
             Collectors.counting() // the downstream collector
          )
       );
```

- Collectors.counting() just counts the number of people of each age

So What Is a Stream?

- **An object that allows one to define processings on data**
  - There is no limit on the amount of data that can be processed
- **Those processings are typically map / filter / reduce operations**

- **Those processings are optimized :**
- **First, we define all the operations**
- **Then, the operations are triggered**

So What Is a Stream?

- Last remark:
- A Stream cannot be « reused »
- Once it has been used ot process a set of data, it cannot be used again to process another set

In Java 8, Stream cannot be reused, once it is consumed or used by a terminal operation, the stream will be closed.

the following example, it will throw an IllegalStateException, saying "stream is closed".

```
public static void main(String[] args) {

    String[] array = {"a", "b", "c", "d", "e"};
    Stream<String> stream = Arrays.stream(array);

    // loop a stream
    stream.forEach(x -> System.out.println(x));

    // reuse it to filter again! throws IllegalStateException

    long count = stream.filter(x -> "b".equals(x)).count();
    System.out.println(count);

}
```

Note: For whatever reason, if you really want to reuse a Stream, try the following Supplier solution :

```
public static void main(String[] args) {

    String[] array = {"a", "b", "c", "d", "e"};

    Supplier<Stream<String>> streamSupplier = () -> Stream.of(array);
```

```
        //get new stream
        streamSupplier.get().forEach(x -> System.out.println(x));

        //get another new stream
        long count = streamSupplier.get().filter(x -> "b".equals(x)).count();

        System.out.println(count);

    }
```

Note: Each get() will return a new stream.

Summary

- **Quick explanation of the map / filter / reduce**
- **What is a Stream**
- **The difference between *intermediary* and *final* operations**
- **The « consuming » operations: forEach() and peek()**
- **The « mapping » operations: map() and flatMap()**
- **The « filter » operation: filter()**
- **The « reduction » operations:**
  - Aggregations: reduce(), max(), min(), …
  - Mutable reductions: collect, Collectors