howest
hogeschool

Object Oriented Architectures and Secure Development

Stream API

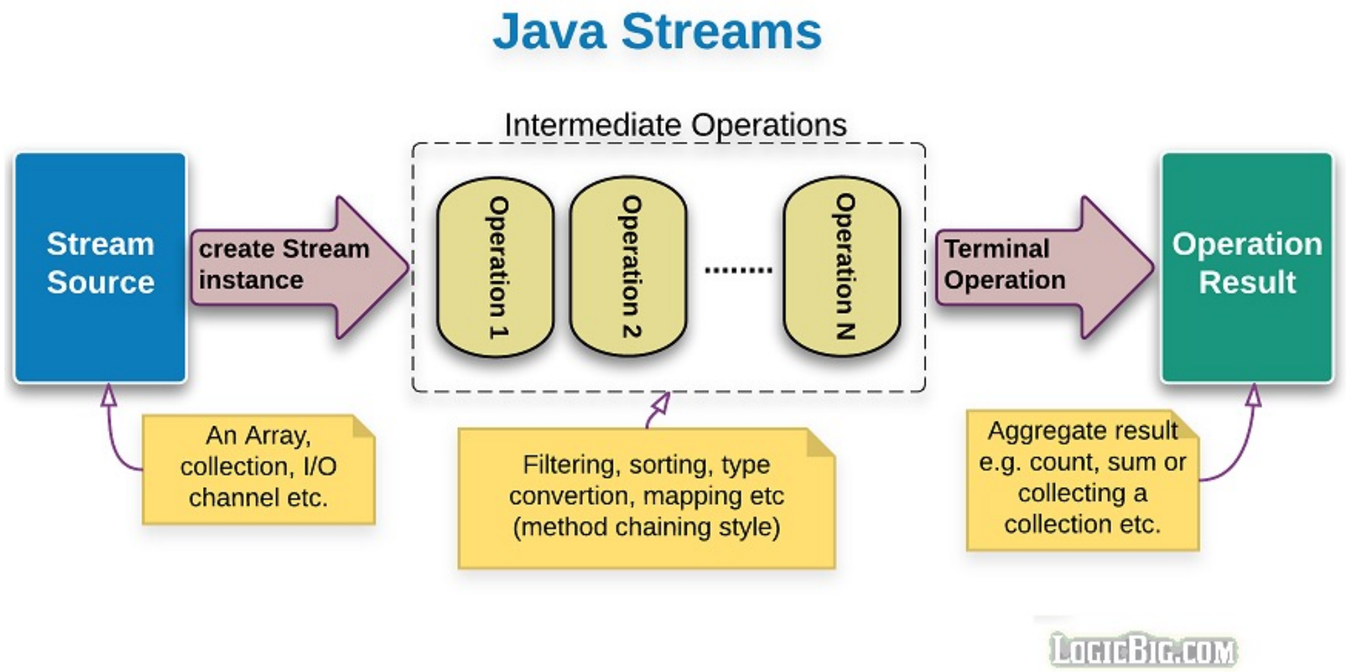
Matthias Blomme
Mattias De Wael
Frédéric Vlummens

Streams ≠ (File) Input/Output

- java.util.stream
- Contains various classes to support functional-style operations on streams of elements. **Nothing to do with file I/O!**
- Examples:
  - map
  - reduce
  - filter
  - forEach
  - …
- But also lots more!

What is a stream?

- Represents a sequence of elements
- Supports various types of operations, allowing for computations on said elements



Example 1: creating a stream from a list of values

```
Stream.of("Jughead", "Betty", "Archie", "Veronica")
        .map(String::toLowerCase)
        .sorted()
        .forEach(System.out::println);
```

```
archie
betty
jughead
veronica
```


howest hogeschool

Example 1: creating a stream from a list of values

```
Stream.of("Jughead", "Betty", "Archie", "Veronica")
        .map(String::toLowerCase)
        .sorted()
        .forEach(System.out::println);
```

- Comparable to JavaScript map function
- Stream (and hence every element in stream) is mapped to another stream
- We provide the method to apply to each element as argument

howest hogeschool

Example 1: creating a stream from a list of values

```
Stream.of("Jughead", "Betty", "Archie", "Veronica")
        .map(String::toLowerCase)
        .sorted()
        .forEach(System.out::println);
```

- The :: operator is also called **method reference operator**
- Pass a reference to methods –in this case– map() and forEach() methods
- Is shorthand for a lambda expression that executes just one method, comparable to:

```
Stream.of("Jughead", "Betty", "Archie", "Veronica")
        .map(e -> e.toLowerCase())
        .sorted()
        .forEach(e -> System.out.println(e));
```


howest hogeschool

Example 2: converting a collection into a stream

```
List<Product> products = new ArrayList<>();
products.add(new Product(1, "cookies", 2.99));
products.add(new Product(2, "chocolate", 3.99));
products.add(new Product(3, "bananas", 1.99));

products.stream()
        .filter(e -> e.getPrice() > 2)
        .sorted(Comparator.comparingDouble(Product::getPrice))
        .forEach(System.out::println);
```

```
Product{id=1, name='cookies', price=2.99}
Product{id=2, name='chocolate', price=3.99}
```


howest hogeschool

Intermediate versus terminal operations

- Intermediate operations
  - Return a stream
  - We can apply chaining → method1().method2().method3();

  - Examples:

    .map()
    .filter()
    .sorted()
    .peek()

```
Stream.of("Jughead", "Betty", "Archie", "Veronica")
        .map(String::toLowerCase)
        .sorted()
        .forEach(System.out::println);
```

Intermediate versus terminal operations

- Terminal operations
  - Return nothing (void) or a non-stream result (int, double, …)
  - Therefore, after a terminal operation, no longer possible to chain

  - Examples:

    .forEach()
    .sum()
    .max()

```
Stream.of("Jughead", "Betty", "Archie", "Veronica")
        .map(String::toLowerCase)
        .sorted()
        .forEach(System.out::println);
```


howest hogeschool

Creating streams

- Stream.of(obj1, obj2, obj3, …)
- IntStream.of(int1, int2, int3, …)
- DoubleStream.of(dbl1, dbl2, dbl3, …)
- LongStream.of(lng1, lng2, lng3, …)
- Collection.stream()

Transforming object streams to primitive streams

```
double avg = products.stream()
        .mapToDouble(e -> e.getPrice())
        .average()
        .orElse(0);

System.out.println(String.format("Average price is %.2f", avg));
```

```
Average price is 2,99
```

Transforming object streams to primitive streams

```
double avg = products.stream()
        .mapToDouble(e -> e.getPrice())
        .average()
        .orElse(0);

System.out.println(String.format("Average price is %.2f", avg));
```

```
Average price is 2,99
```

- average() returns an OptionalDouble
- Here, calling orElse(0) will return the double inside or 0 if there is none.
- Other useful method of OptionalXXX: isPresent (boolean)

Transforming primitive streams to object streams

```
IntStream.range(1, 4)
         .mapToObj(e -> new Product(e, "Prod " + e, e*2))
         .forEach(System.out::println);
```

```
Product{id=1, name='Prod 1', price=2.0}
Product{id=2, name='Prod 2', price=4.0}
Product{id=3, name='Prod 3', price=6.0}
```


howest hogeschool

Laziness

```
Stream.of("Jughead", "Betty", "Archie", "Veronica")
        .map(e -> {
            System.out.println(e);
            return e.toUpperCase();
        });
```

- Nothing gets printed.
- Intermediate operations are only evaluated if a terminal operation is present.

Execution order

```
Stream.of("Jughead", "Betty", "Archie", "Veronica")
        .map(e -> {
            System.out.println(e);
            return e.toUpperCase();
        })
        .forEach(System.out::println);
```

```
Jughead
JUGHEAD
Betty
BETTY
Archie
ARCHIE
Veronica
VERONICA
```

- You might think the Stream would move horizontally and do all map operation first and then the forEach operations.
- This is not the case: each element moves along the chain vertically, as is demonstrated by this snippet's output.

howest hogeschool

Efficiency

- Think about the order of your chain.
- Compare this snippet:

```
Stream.of("Jughead", "Betty", "Archie", "Veronica")
        .map(String::toUpperCase)
        .filter(e -> e.contains("C"))
        .forEach(System.out::println);
```

- To this snippet:

```
Stream.of("Jughead", "Betty", "Archie", "Veronica")
        .filter(e -> e.contains("c"))
        .map(String::toUpperCase)
        .forEach(System.out::println);
```

Collect

- Very useful terminal operation.
- Allows you to transform elements of a stream into a different result, such as List, Set or Map.

```
List<Person> persons = new ArrayList<>();
persons.add(new Person("Jughead", 18));
persons.add(new Person("Betty", 18));
persons.add(new Person("Veronica", 21));
persons.add(new Person("Archie", 20));

List<Person> filteredPersons =
    persons.stream()
           .filter(p -> p.getAge() > 18)
           .collect(Collectors.toList());
```


howest hogeschool

Collect – more examples

```
Map<Integer, List<Person>> personsPerAge =
        persons.stream()
       .collect(Collectors.groupingBy(Person::getAge));

personsPerAge.forEach((a, p) -> System.out.println(
        String.format("%s %s", a, p)));
```

```
18 [Person{name='Jughead', age=18}, Person{name='Betty', age=18}]
20 [Person{name='Archie', age=20}]
21 [Person{name='Veronica', age=21}]
```

Collect – more examples

```
double avgAge =
        persons.stream()
               .collect(Collectors.averagingDouble(Person::getAge));
System.out.println(avgAge);
```

```
19.25
```


howest hogeschool

Collect – more examples

```
DoubleSummaryStatistics stats =
    persons.stream()
           .collect(Collectors.summarizingDouble(Person::getAge));

System.out.println(stats);
```

```
DoubleSummaryStatistics{count=4, sum=77,000000,
min=18,000000, average=19,250000, max=21,000000}
```

Useful references

- https://www.baeldung.com/java-streams