Programming in Java
Part VI – Streams

Carlos Kavka

Head of Research and Development

Agenda

Streams

Specialized streams

Optional

Advanced stream operations

Collectors

Streams

represent a sequence of elements on which operations can be applied

supports internal iteration

streams

can be used to traverse collections

traversable only once

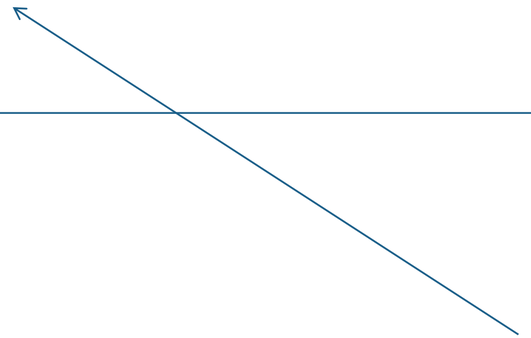
potentially unlimited in size

A first example

```
Stream<String> s1 = Stream.of("Mariapia", "Enrico", "Stefano");
Stream<Integer> s2 = Stream.of(12, 34, 55);

s1.filter(x -> x.length() > 6)
    .forEach(System.out::println);

s2.forEach(System.out::println);
```



internal iteration

Traversable only once

This is OK!

```
Stream<Integer> s1 = Stream.of(12, 34, 55);
Stream s2 = s1.filter(x -> x > 30);
s2.forEach(System.out::println);
```

This is not OK!!!!

```
Stream<Integer> s1 = Stream.of(12, 34, 55);
s1.filter(x -> x > 30);
s1.forEach(System.out::println);
```

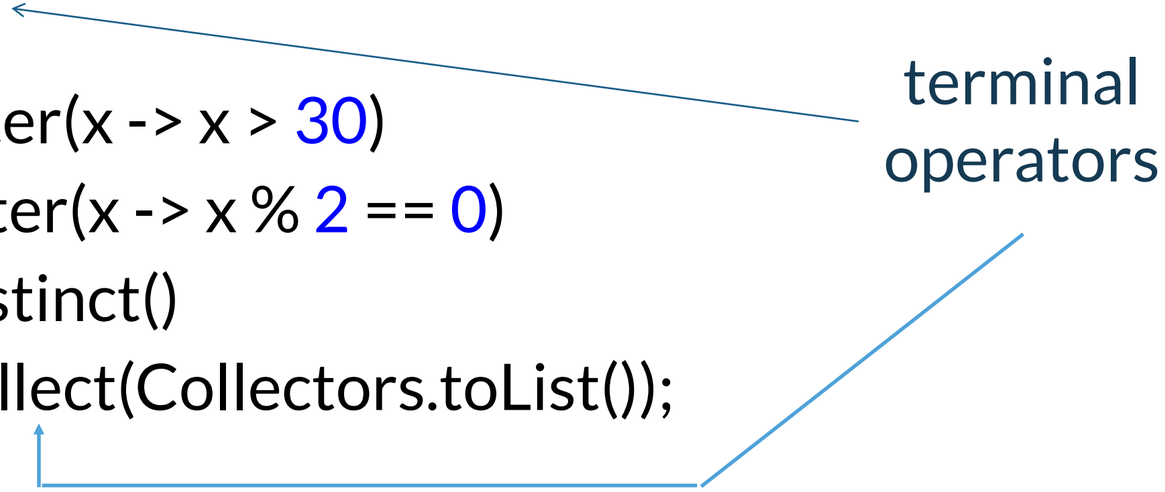
Terminal and intermediate operators

```
Stream<String> s1 = Stream.of("Mariapia", "Enrico", "Stefano");
Stream<Integer> s2 = Stream.of(12, 34, 55);

System.out.println(s1.count());

List<Integer> list1 = s2.filter(x -> x > 30)
                        .filter(x -> x % 2 == 0)
                        .distinct()
                        .collect(Collectors.toList());

list1.forEach(System.out::println);
```



Execution starts with the terminal operator

Terminal and intermediate operators

```
Stream<String> s1 = Stream.of("Mariapia", "Enrico", "Stefano");

List<String> list1 = s1.filter(x -> x.startsWith("M"))
                                .collect(Collectors.toList());

list1.forEach(System.out::println);
```

Other intermediate operators

```
Stream<String> s1 = Stream.of("Mariapia", "Enrico", "Stefano");

List<String> list1 = s1.filter(x -> x.length() > 2)
                       .limit(2)
                       .skip(1)
                       .collect(Collectors.toList());

list1.forEach(System.out::println);
```

Optional values

```
Stream<Integer> s2 = Stream.of(12, 34, 55);

Optional<Integer> value = s2.filter(x -> x > 30)
                            .filter(x -> x % 2 == 0)
                            .findFirst();

value.ifPresent(System.out::println);
```

Please note that findFirst() returns an optional. Why?

Optional values

```
Stream<Integer> s2 = Stream.of(12, 34, 55);

Optional<Integer> value = s2.filter(x -> x > 30)
                            .filter(x -> x % 2 == 0)
                            .findAny();

value.ifPresent(System.out::println);
```

findAny() is better for parallel execution. Why?

Other final operators

```
Stream<Integer> s1 = Stream.of(12, 34, 55);
boolean value1 = s1.allMatch(x -> x > 2);

Stream<Integer> s2 = Stream.of(12, 34, 55);
boolean value2 = s2.anyMatch(x -> x > 2);

Stream<Integer> s3 = Stream.of(12, 34, 55);
boolean value3 = s3.noneMatch(x -> x > 2);
```

these operators implement short-circuit behavior

Mapping

Convert all strings to lower case

```
Stream<String> s1 = Stream.of("Mariapia", "Enrico", "Stefano");

List<String> list1 = s1.map(String::toLowerCase)
                       .collect(Collectors.toList());
```

Mapping

Get the first character of all strings

```
Stream<String> s1 = Stream.of("Mariapia", "Enrico", "Stefano");

List<Character> list1 = s1.map(x -> x.charAt(0))
                          .collect(Collectors.toList());
```

Mapping

Get the length of all strings

```
Stream<String> s1 = Stream.of("Mariapia", "Enrico", "Stefano");

List<Integer> list1 = s1.map(x -> x.length())
                        .collect(Collectors.toList());
```

Mapping

All strings in lowercase in alphabetical order

```
Stream<String> s1 = Stream.of("Mariapia", "Enrico", "Stefano");

List<String> list1 = s1.map(String::toLowerCase)
                       .sorted()
                       .collect(Collectors.toList());
```

Mapping

All strings in lowercase sorted by length

```
Stream<String> s1 = Stream.of("Mariapia", "Enrico", "Stefano");

List<String> list1 = s1.map(String::toLowerCase)
                       .sorted(Comparator.comparing(String::length))
                       .collect(Collectors.toList());
```

Mapping

All surnames in lower case

```
Stream<String> s1 = Stream.of("Nina Pušlar", "Jan Plestenjak", "Tinkara Kovač");

List<String> list1 = s1.map(String::toLowerCase)
                       .sorted(Comparator
                               .comparing((Function<String, String>)
                                          (x -> x.substring(x.indexOf(" ")))))
                       .collect(Collectors.toList());
```

Creating a stream

many possibilities

```
List<String> list1 = Arrays.asList("Stefano", "Mariapia", "Enrico");
List<String> list2 = Arrays.asList(("Nina", "Jan", "Tinkara");

list1.stream().count();

HashSet<String> set1 = new HashSet<>();
set1.stream().count();

Stream s1 = Stream.empty();
Stream s2 = Stream.of(list1, list2);
```

Numeric streams

created with range() and rangeClosed()

```
IntStream ints1 = IntStream.range(0, 10);
System.out.println(
      ints1.filter(x -> x % 2 == 0).count()
);

IntStream ints2 = IntStream.rangeClosed(0, 10);
System.out.println(
      ints2.filter(x -> x % 2 == 0).count()
);
```

Numeric streams

created by mapping from other streams

```
Stream<Integer> s1 = Stream.of(12, 34, 34, 55, 102);
System.out.println(
      s1.mapToInt(x -> x + 1).sum()
);

Stream<Integer> s2 = Stream.of(12, 34, 34, 55, 102);
System.out.println(
      s2.mapToInt(x -> x + 1).max()
);
```

however…

Numeric streams

be careful with max() and min() !

```
Stream<Integer> s2 = Stream.of(12, 34, 34, 55, 102);

OptionalInt value = s2.mapToInt(x -> x + 1).min();
value.ifPresent(System.out::println);
```

Note the specialized Optional definition

Other stream creation options

```
int [] a = {1, 2, 3};
System.out.println(Arrays.stream(a).sum());

LongStream st1 = LongStream.iterate(2, x -> x * x);
long []b = st1.limit(5).toArray();

Arrays.stream(b).forEach(System.out::println);

Stream.generate(Math::random)
      .limit(5)
      .forEach(System.out::println);
```

What about the length of these streams?

The flatMap operator

```
int []c = IntStream.rangeClosed(0, 2)
                        .flatMap(x -> IntStream.rangeClosed(0, 2)
                                                .map(y -> x + y))
                        .toArray();

List<Integer> m2 = Stream.of(Arrays.asList(1,2,3), Arrays.asList(4,5,6))
                                .flatMap(x -> x.stream())
                                .collect(Collectors.toList());
```

peek()

Stream<T> **peek**(Consumer<? super T> action)

produces a stream after applying the operation

only for debugging!

```
OptionalInt value = IntStream.of(1, 2, 3, 4)
    .peek(x -> System.out.println("processing: " + x))
    .filter(x -> x % 2 == 0)
    .peek(x -> System.out.println("accepted " + x))
    .findFirst();
```

Other map flavors

produces a stream of primitive types

DoubleStream **mapToDouble**(ToDoubleFunction<? super T> mapper)
IntStream **mapToInt**(ToIntFunction<? super T> mapper)
LongStream **mapToLong**(ToLongFunction<? super T> mapper)

```
List<String> list6 = Arrays.asList("Mariapia", "Teresa");

int sum = list6.stream()
                .mapToInt(String::length)
                .sum()
```

Other map flavors

can change the type of a stream of primitive types

IntStream **map**(IntUnaryOperator mapper)
DoubleStream **mapToDouble**(IntToDoubleFunction mapper)
LongStream **mapToLong**(IntToLongFunction mapper)
Stream<T> **mapToObj**(IntFunction<? extends T> mapper)

```
List<Integer> list7 = IntStream.rangeClosed(1, 10)
        .mapToObj(x -> x * 2)
        .collect(Collectors.toList());
```

boxed()

converts a specialized stream into a Stream with boxed values

```
List<Integer> list8 = IntStream
        .rangeClosed(1, 10)
        .boxed()
        .collect(Collectors.toList());
```

unordered(), parallel() and sequential()

unordered() transforms the stream from sequential to unordered

parallel() determines a parallel mode for execution of the stream

sequential() determines a sequential mode for execution of the stream

unordered(), parallel() and sequential()

parallel processing example

```
List<Integer> list8 = IntStream.rangeClosed(1, 10)
      .boxed()
      .collect(Collectors.toList());

List<Integer> list9 = list8.stream()
      .unordered()
      .parallel()
      .peek(x -> System.out.println(Thread.currentThread()
                                          .getName()))
      .map(x -> x + 1)
      .collect(Collectors.toList());
```

unordered(), parallel() and sequential()

what happens here?

```
List<Integer> list8 = IntStream.rangeClosed(1, 10)
      .boxed()
      .collect(Collectors.toList());

List<Integer> list9 = list8.stream()
      .unordered()
      .parallel()
      .peek(x -> System.out.println(Thread.currentThread()
                                          .getName()))
      .sequential()
      .map(x -> x + 1)
      .collect(Collectors.toList());
```

unordered(), parallel() and sequential()

the stream has a single
execution mode!

forEachOrdered()

processes the elements in the order specified by the stream,
independently if the stream is executed serial or parallel

```
IntStream.rangeClosed(1, 100)
        .parallel()
        .map(x -> x + 1)
        .forEachOrdered(System.out::println);
```

A bit more about flatMap()

these two examples are equivalent

```
List<String> list13 = Arrays.asList("Mariapia", "Teresa");

list13.stream()
      .map(x -> x.length())
      .forEachOrdered(System.out::println);

list13.stream()
      .flatMap(x -> Stream.of(x.length()))
      .forEachOrdered(System.out::println);
```

A bit more about flatMap()

get, for each number *x* in the input stream,
the pair (*x*, 2**x*)

```
List<Integer> list8 = IntStream.rangeClosed(1, 10)
        .boxed()
        .collect(Collectors.toList());

list8.stream()
        .map(x -> new int[]{x, 2 * x})
        .forEach(x -> System.out.println(x[0] + ", " + x[1]));
```

A bit more about flatMap()

it can also be implemented as

```
list8.stream()
      .flatMap(x -> Stream.of(x, 2 * x))
      .forEach(System.out::println);
```

or even better

```
IntStream.rangeClosed(1, 10)
      .flatMap(x -> IntStream.of(x, 2 * x))
      .forEach(System.out::println);
```

A bit more about flatMap()

create a single stream from two lists

```
Stream.of(list11, list12)
      .flatMap(x -> x.stream())

.forEachOrdered(System.out::println);
```

A bit more about flatMap()

combining values from two streams

```
list11.stream()
      .flatMap(x -> list12.stream()
            .flatMap(y -> Stream.of(x, y)))
      .forEachOrdered(x -> System.out.print(x + " "));
```

reduce()

combine the elements of a stream
repeatedly to produce a single value

summation

```
int tot = list15.stream()
                .reduce(0, (x, y) -> x + y);
```

product

```
int tot = list15.stream()
                .reduce(1, (x, y) -> x * y);
```

reduce()

it can be also written as

```
int tot3 = list15.stream()
                 .reduce(0, Integer::sum);
```

note that the initial value can be omitted

```
Optional<Integer> tot4 = list15.stream()
                              .reduce((x,y) -> x + y);
```

reduce()

calculate the minimum

```
Optional<Integer> tot5 = list15.stream()
                                .reduce((x, y) -> x < y ? x : y);
```

other possibility

```
Optional<Integer> tot6 = list15.stream()
                                .reduce(Integer::min);
```

reduce()

what about concatenation of strings?

```
List<String> list16 = Arrays.asList("Stefano", "Mariapia", "Enrico");
String str = list16.stream().reduce("", (x,y) -> x + y);
```

other possibility:

```
String str2 = books
        .stream()
        .collect(Collectors
                .reducing("titles: ", Book::getTitle, (x, y) -> x + y));
```

reduce

other examples

```
int count = books
      .stream()
      .map(x -> 1)
      .reduce(0, (x,y) -> x + y);
```

```
int totalPages = books
      .stream()
      .collect(Collectors
            .reducing(0, Book::getNumberOfPages, (x,y) -> x + y));
```

Thank you!

esteco.com