



Lesson 17

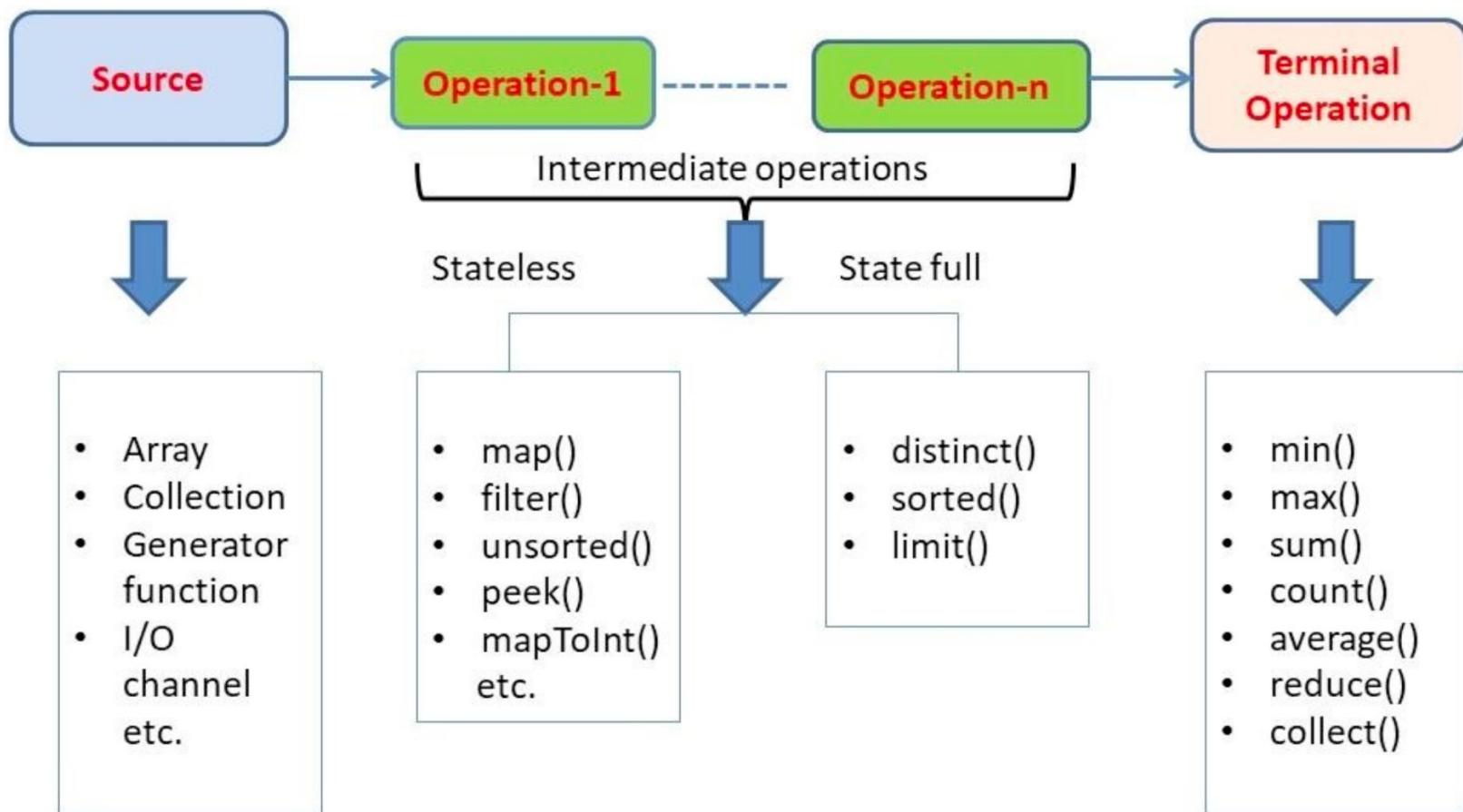
12.09.2024

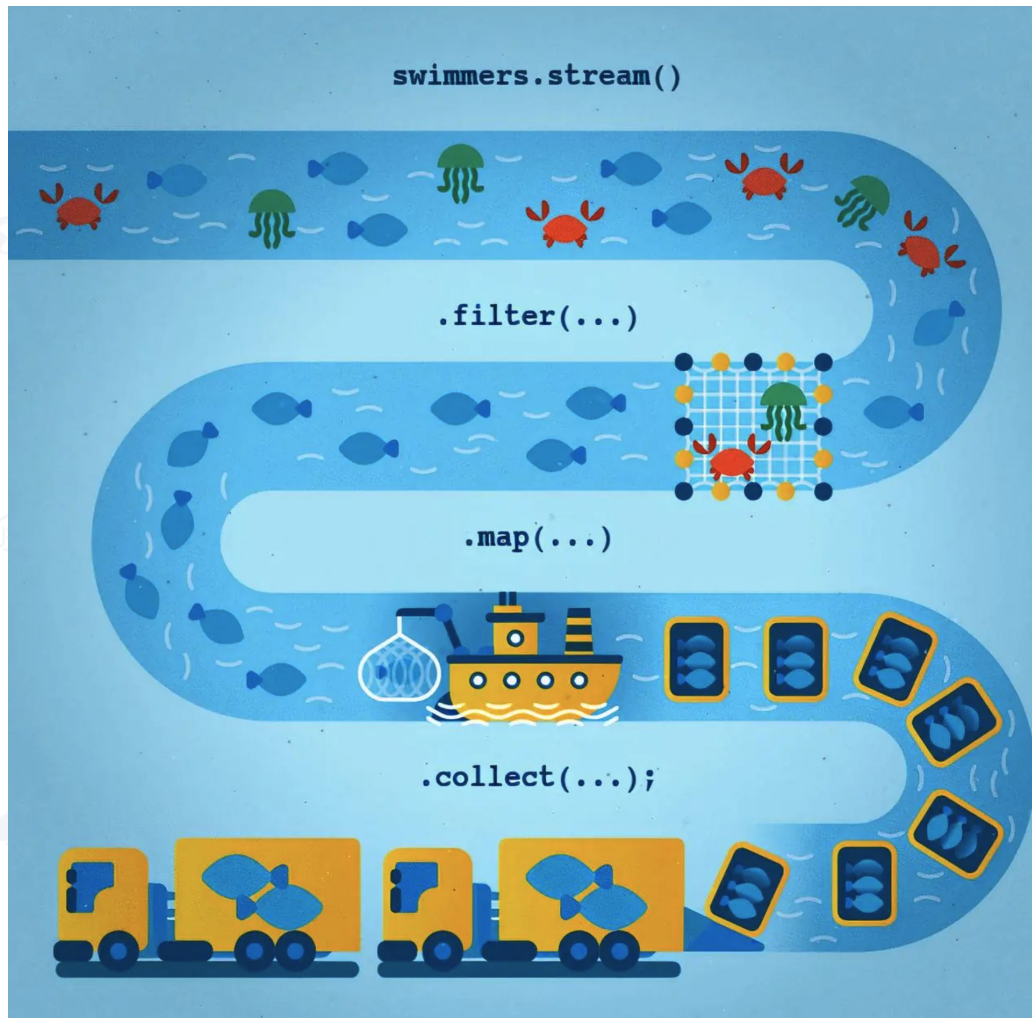
```
public class Ex1 {  
    public static void main(String[] args) {  
        Integer i = 10;  
        List<Integer> list = new ArrayList<>();  
        list.add(i);  
        list.add(i *= 2);  
        list.add(i);  
        list.removeIf(j -> j == 10);  
        System.out.println(list);  
    }  
}
```

```
public class Ex2 {  
    public static void main(String[] args) {  
        List<String> trafficLight = new ArrayList<>();  
        trafficLight.add("RED");  
        trafficLight.add(1, "ORANGE");  
        trafficLight.add(2, "GREEN");  
        trafficLight.remove(Integer.valueOf(2));  
        System.out.println(trafficLight);  
    }  
}
```

```
public class Ex3 {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<>();  
        list.add("meat");  
        list.add("bread");  
        list.add("sassage");  
        Stream<String> stream = list.stream()  
                                .filter(a -> a.length() < 5)  
                                .map(a -> a + "_map");  
  
        list.add("eggs");  
        stream.forEach(System.out::println);  
    }  
}
```

```
public class Ex4 {  
    public static void main(String[] args) {  
        Set<String> set = new TreeSet<>();  
        List<String> list = Stream.of("JPoint",  
                                     "HolyJS",  
                                     "Devovx",  
                                     "Devovx",  
                                     "HolyJS",  
                                     "JPoint")  
                                .sequential()  
                                .filter(set::add)  
                                .peek(System.out::println)  
                                .collect(Collectors.toList());  
        System.out.println(list);  
    }  
}
```



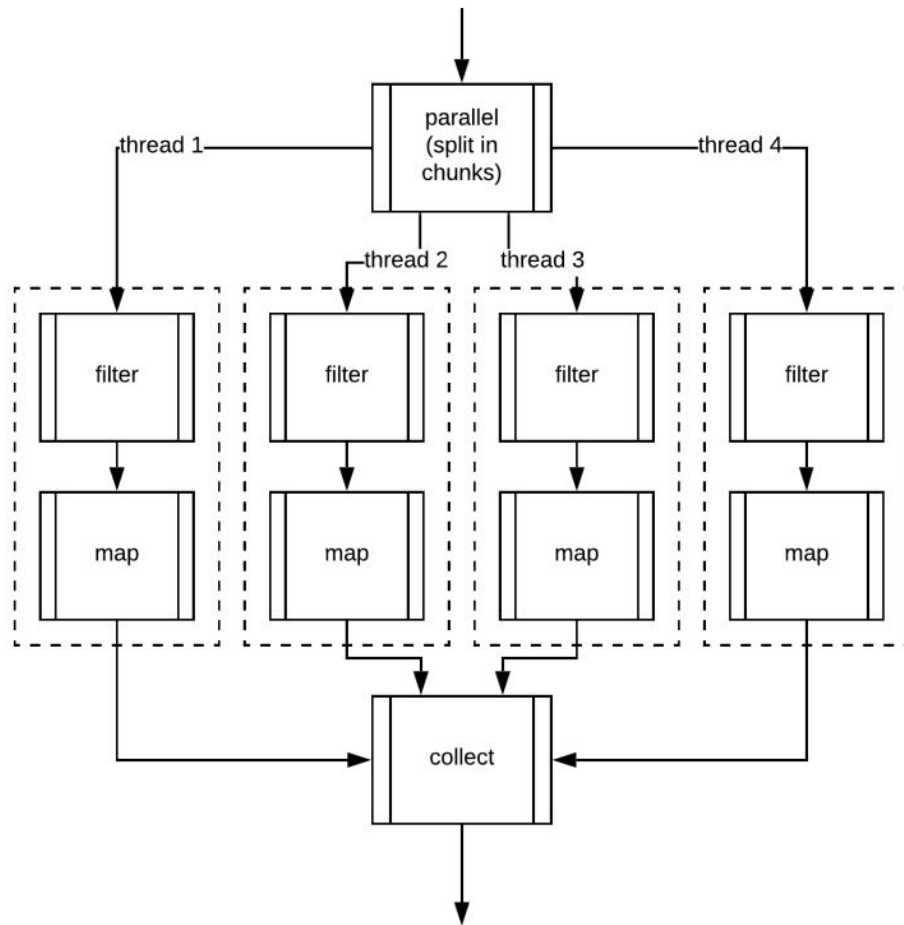


Получение объекта Stream

Пока что хватит теории. Пришло время посмотреть, как создать или получить объект `java.util.stream.Stream`.

- Пустой стрим: `Stream.empty()`
 - Стрим из List: `list.stream()`
 - Стрим из Map: `map.entrySet().stream()`
 - Стрим из массива: `Arrays.stream(array)`
 - Стрим из указанных элементов: `Stream.of("a", "b")`
- ```
// Stream<String>
// Stream<String>
// Stream<Map.Entry<String, String>>
// Stream<String>
// Stream<String>
```







## **void forEach(Consumer action)**

Виконує вказану дію для кожного елемента стримування.

```
Stream.of(120, 410, 85, 32, 314, 12)
 .forEach(x -> System.out.format("%s, ", x));
```

## void forEachOrdered(Consumer action)

Теж виконує вказану дію для кожного елемента стриму, але домагається правильного порядку входження елементів. Використовується для паралельних стримувань, коли потрібно отримати правильну послідовність елементів.

```
IntStream.range(0, 100000)
 .parallel()
 .filter(x -> x % 10000 == 0)
 .map(x -> x / 10000)
 .forEachOrdered(System.out::println);
```

## R collect(Collector collector)

Один із найпотужніших операторів Stream API. З його допомогою можна зібрати всі елементи в список, безліч або іншу колекцію, згрупувати елементи за якимось критерієм, об'єднати все в рядок і т.д.. У класі `java.util.stream.Collectors` дуже багато методів на всі випадки життя, ми розглянемо їх пізніше. За бажання можна написати свій колектор, реалізувавши інтерфейс `Collector`.

```
List<Integer> list = Stream.of(1, 2, 3)
 .collect(Collectors.toList());
```

```
String s = Stream.of(1, 2, 3)
 .map(String::valueOf)
 .collect(Collectors.joining("-", "<", ">"));
```



Optional min(Comparator comparator)  
Optional max(Comparator comparator)

```
int min = Stream.of(20, 11, 45, 78, 13)
 .min(Integer::compare).get();
```

```
int max = Stream.of(20, 11, 45, 78, 13)
 .max(Integer::compare).get();
```

## Optional findAny()

Повертає перший елемент стриму, що попався. У паралельних стримах це може бути справді будь-який елемент, який лежав у будь-якій частині послідовності.

## Optional findFirst()

Гарантовано повертає перший елемент стриму, навіть якщо стрім паралельний

## boolean allMatch(Predicate predicate)

Повертає true, якщо всі елементи стриму задовольняють умову predicate. Якщо зустрічається якийсь елемент, для якого результат виклику функції-предикату буде false, оператор перестає переглядати елементи і повертає false.

```
boolean result = Stream.of(1, 2, 3, 4, 5)
 .allMatch(x -> x <= 7);
```

### **boolean anyMatch(Predicate predicate)**

Повертає true, якщо хоч один елемент стриму задовольняє умові predicate. Якщо такий елемент зустрівся, немає сенсу продовжувати перебір елементів, тому одразу повертається результат.

```
boolean result = Stream.of(1, 2, 3, 4, 5)
 .anyMatch(x -> x == 3);
```

### **boolean noneMatch(Predicate predicate)**

Повертає true, якщо, пройшовши всі елементи стриму, жоден не задовольнив умову predicate. Якщо зустрічається якийсь елемент, для якого результат виклику функції-предикату буде true, оператор перестає перебирати елементи і повертає false.

```
boolean result = Stream.of(120, 410, 86, 32, 314, 12)
 .noneMatch(x -> x % 2 == 1);
```



## OptionalDouble average()


Тільки для примітивних стримувань. Повертає середнє арифметичне всіх елементів. Або Optional.empty, якщо стриманий порожній.

```
double result = IntStream.range(2, 16)
 .average()
 .getAsDouble();
```




## sum()

Повертає суму елементів примітивного стримування. Для IntStream результат буде типу int, для LongStream – long, для DoubleStream – double.



```
long result = LongStream.range(2, 16)
 .sum();
```



## toList()

Найпоширеніший метод. Збирає елементи у List.

## toSet()

Збирає елементи у множину.

## toMap(Function keyMapper, Function valueMapper)

Збирає елементи в Map. Кожен елемент перетворюється на ключ і значення, ґрунтуючись на результаті функцій keyMapper і valueMapper відповідно

```
Map<Character, String> map3 = Stream.of(50, 54, 55)
 .collect(Collectors.toMap(
 i -> (char) i.intValue(),
 i -> String.format("<%d>", i)
));
```



counting()

```
Long count = Stream.of("1", "2", "3", "4")
 .collect(Collectors.counting());
System.out.println(count);
```

**minBy(Comparator comparator)**

**maxBy(Comparator comparator)**

```
Optional<String> min = Stream.of("ab", "c", "defgh", "ijk", "l")
 .collect(Collectors.minBy(Comparator.comparing(String::length)));
min.ifPresent(System.out::println);
```

```
Optional<String> max = Stream.of("ab", "c", "defgh", "ijk", "l")
 .collect(Collectors.maxBy(Comparator.comparing(String::length)));
max.ifPresent(System.out::println);
```



**groupBy(Function classifier)**

**groupBy(Function classifier, Collector downstream)**

**groupBy(Function classifier, Supplier mapFactory, Collector downstream)**

```
Map<Integer, List<String>> map1 = Stream.of(
 "ab", "c", "def", "gh", "ijk", "l", "mnop")
 .collect(Collectors.groupingBy(String::length));
map1.entrySet().forEach(System.out::println);
```