Lesson 14

13.05.2021

```
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",
                "Devoxx",
                "Devoxx",
                "HolyJS",
                "JPoint")
                .sequential()
                .filter(set::add)
                .peek(System.out::println)
                .collect(Collectors.toList());
        System.out.println(list);
    }
}
```

Лямбда представляет собой набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы.

Основу лямбда-выражения составляет лямбда-оператор, который представляет стрелку ->. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая собственно представляет тело лямбда-выражения, где выполняются все действия.

```
(x, y) -> x + y;
```

Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в функциональном интерфейсе. При этом важно, что функциональный интерфейс должен содержать только один единственный метод без реализации.

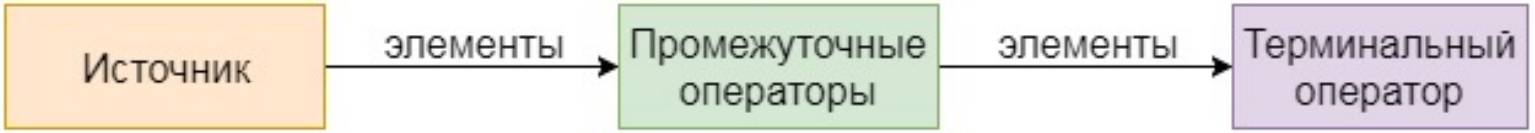
По факту лямбда-выражения являются в некотором роде сокращенной формой внутренних анонимных классов, которые ранее применялись в Java.

- Отложенное выполнение (deferred execution) лямбда-выражения- определяется один раз в одном месте программы, вызываются при необходимости, любое количество раз и в произвольном месте программы.
- Параметры лямбда-выражения должны соответствовать по типу параметрам метода функционального интерфейса.
- *Конечные лямбда-выражения* не обязаны возвращать какое-либо значение.
- *Блочные лямбда-выражения* обрамляются фигурными скобками. В блочных лямбда-выражениях можно использовать внутренние вложенные блоки, циклы, конструкции if, switch, создавать переменные и т.д. Если блочное лямбда-выражение должно возвращать значение, то явным образом применяется оператор return.
- *Передача лямбда-выражения в качестве параметра метода*.

Stream — это объект для универсальной работы с данными. Мы указываем, какие операции хотим провести, при этом не заботясь о деталях реализации. Например, *взять элементы из списка сотрудников, выбрать тех, кто младше 40 лет, отсортировать по фамилии и поместить в новый список.* Или чуть сложнее, *прочитать все json-файлы, находящиеся в папке books, десериализировать в список объектов книг, обработать элементы всех этих списков, а затем сгруппировать книги по автору*.

Данные могут быть получены из источников, коими являются коллекции или методы, поставляющие данные. Например, список файлов, массив строк, метод range() для числовых промежутков и т.д. То есть, стрим использует существующие коллекции для получения новых элементов, это ни в коем случае не новая структура данных.

К данным затем применяются операторы. Например, взять лишь некоторые элементы (filter), преобразовать каждый элемент (map), посчитать сумму элементов или объединить всё в один объект



Операторы можно разделить на две группы:

- Промежуточные (intermediate) — обрабатывают поступающие элементы и возвращают стрим. Промежуточных операторов в цепочке обработки элементов может быть много.

- Терминальные (terminal) — обрабатывают элементы и завершают работу стрима, так что терминальный оператор в цепочке может быть только один.

Получение объекта Stream

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

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

void forEach(Consumer action)
Выполняет указанное действие для каждого элемента стрима.

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

void forEachOrdered(Consumer action)

Тоже выполняет указанное действие для каждого элемента стрима, но перед этим добивается правильного порядка вхождения элементов. Используется для параллельных стримов, когда нужно получить правильную последовательность элементов.

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

R collect(Collector collector)

Один из самых мощных операторов Stream API. С его помощью можно собрать все элементы в список, множество или другую коллекцию, сгруппировать элементы по какому-нибудь критерию, объединить всё в строку и т.д.. В классе java.util.stream.Collectors очень много методов на все случаи жизни, мы рассмотрим их позже. При желании можно [написать свой коллектор], реализовав интерфейс 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)
Поиск минимального/максимального элемента, основываясь на переданном компараторе.

```
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()

Возвращает первый попавшийся элемент стрима. В параллельных стримах это может быть действительно любой элемент, который лежал в разбитой части последовательности.

Optional findFirst()

Гарантированно возвращает первый элемент стрима, даже если стрим параллельный.

boolean allMatch(Predicate predicate)

Возвращает true, если все элементы стрима удовлетворяют условию predicate. Если встречается какой-либо элемент, для которого результат вызова функции-предиката будет false, то оператор перестаёт просматривать элементы и возвращает false.

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

boolean anyMatch(Predicate predicate)

Возвращает true, если хотя бы один элемент стрима удовлетворяет условию predicate. Если такой элемент встретился, нет смысла продолжать перебор элементов, поэтому сразу возвращается результат.

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

boolean noneMatch(Predicate predicate)

Возвращает true, если, пройдя все элементы стрима, ни один не удовлетворил условию predicate. Если встречается какой-либо элемент, для которого результат вызова функции-предиката будет true, то оператор перестаёт перебирать элементы и возвращает false.

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

OptionalDouble average()

Только для примитивных стримов. Возвращает среднее арифметическое всех элементов. Либо Optional.empty, если стрим пуст.

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

sum()

Возвращает сумму элементов примитивного стрима. Для IntStream результат будет типа int, для LongStream — long, для DoubleStream — double.

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

toList()
Самый распространённый метод. Собирает элементы в List.

toSet()
Собирает элементы в множество.

toMap(Function keyMapper, Function valueMapper)
Собирает элементы в Map. Каждый элемент преобразовывается в ключ и в значение, основываясь на результате функций keyMapper и valueMapper соответственно.

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

counting()

Подсчитывает количество элементов.

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

minBy(Comparator comparator)
maxBy(Comparator comparator)

Поиск минимального/максимального элемента, основываясь на заданном компараторе.

```
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);
```

groupingBy(Function classifier)
groupingBy(Function classifier, Collector downstream)
groupingBy(Function classifier, Supplier mapFactory, Collector downstream)

Группирует элементы по критерию, сохраняя результат в Map. Вместе с представленными выше агрегирующими коллекторами, позволяет гибко собирать данные.

```
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);
```