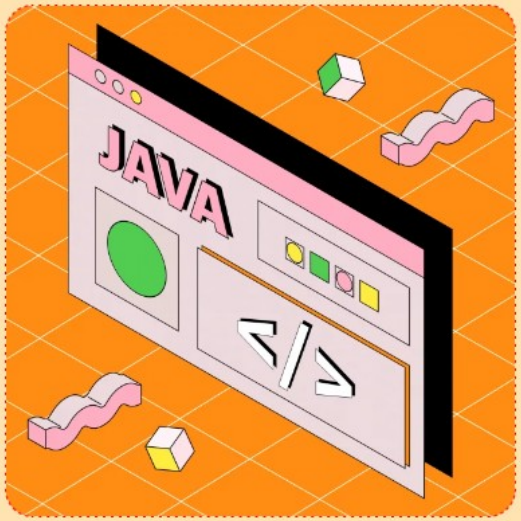
Вебинар

Stream API в Java

- Декларативный и императивный подход
- Практика с использовании стримов



17 сентября 2021 18:00

Спикер: Шибков Константин

образовательная платформа

Skillbox

Декларативный и императивный подход

Skillbox

Императивный подход

Подробное объяснение процесса получения результата:

- Двигайся на автомобиле 10 км на север, выполни крутой поворот на право, прямо проедь до моста, справа от вас будет ул. Релизная, дом 17

- Сравни возвраты двух первых друзей, у кого возвраст больше сравни в третьим. После второго сравнения, полученное значение будет максимальным среди троих.

- Для получения суммы всех неотрицательных значений в списке, получи каждый элемент списка, сравни с 0 – если больше 0 добавь в переменную sum значение. Продолжай до того пока в списке будут элементы.

Декларативный подход

Требование результата, как его получить не объясняется:

- Такси до ул. Релизной, дом 17
- У меня есть три числа – найдите максимальное среди них
- Верните сумму чисел списка, исключая отрицательные.

Но кто-то должен знать как действия выполнять для достижения результата?

Декларативные подход содержим в себе Императивный, для реализации задачи

Давайте пример в коде

Объясняем как делать

```
List<Integer> integerList =
    List.of(1, 2, 3, -1, 5, -5);

int sum = 0;
for (int i : integerList) {
    if(i > 0){
        sum+=i;
    }
}

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

Пишем, что хотим получить

```
List<Integer> integerList =
    List.of(1, 2, 3, -1, 5, -5);

int sum = integerList.stream()
    .mapToInt(Integer::intValue)
    .filter(i -> i > 0)
    .sum();

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

Еще один

Из списка строк, получить одну строку, в которой через запятую будут записаны длины строк в списке

Объясняем как делать

```
List<String> stringList =
    List.of("se", "n", "del", "ru");

String result = "";
for (int i = 0; i < stringList.size(); i++)
{
    result += stringList.get(i).length();
    if (i < stringList.size() - 1) {
        result += ",";
    }
}

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

Пишем, что хотим получить

```
List<String> stringList =
    List.of("se", "n", "del", "ru");

String result = stringList.stream()
    .map(String::length)
    .map(String::valueOf)
    .collect(Collectors.joining(","));

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

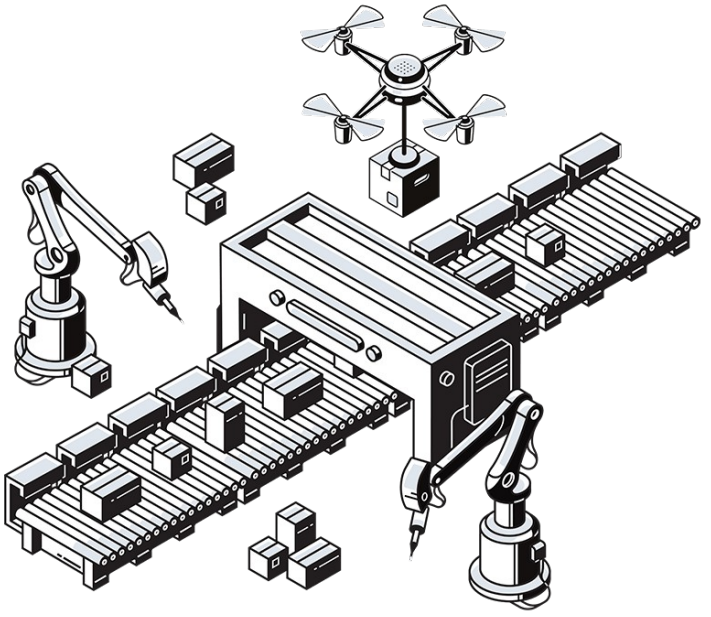
Почему бы и нет

Пишем, что хотим получить

```
List<String> stringList =
    List.of("se", "n", "del", "ru");

String result = new MyMagicClass().makeAwesome(stringList);

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

Stream API

минимум теории – максимум практики



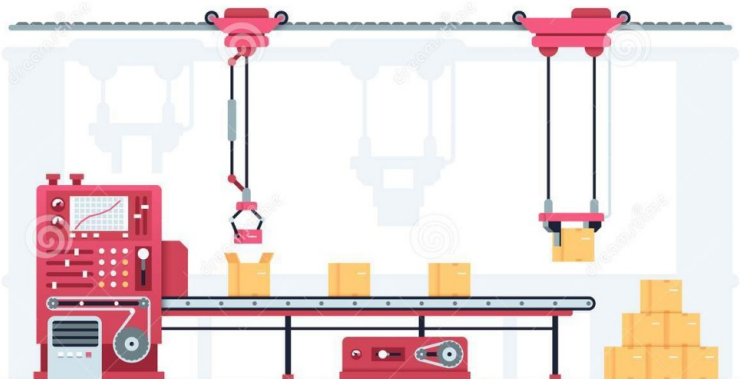
Skillbox

StreamAPI

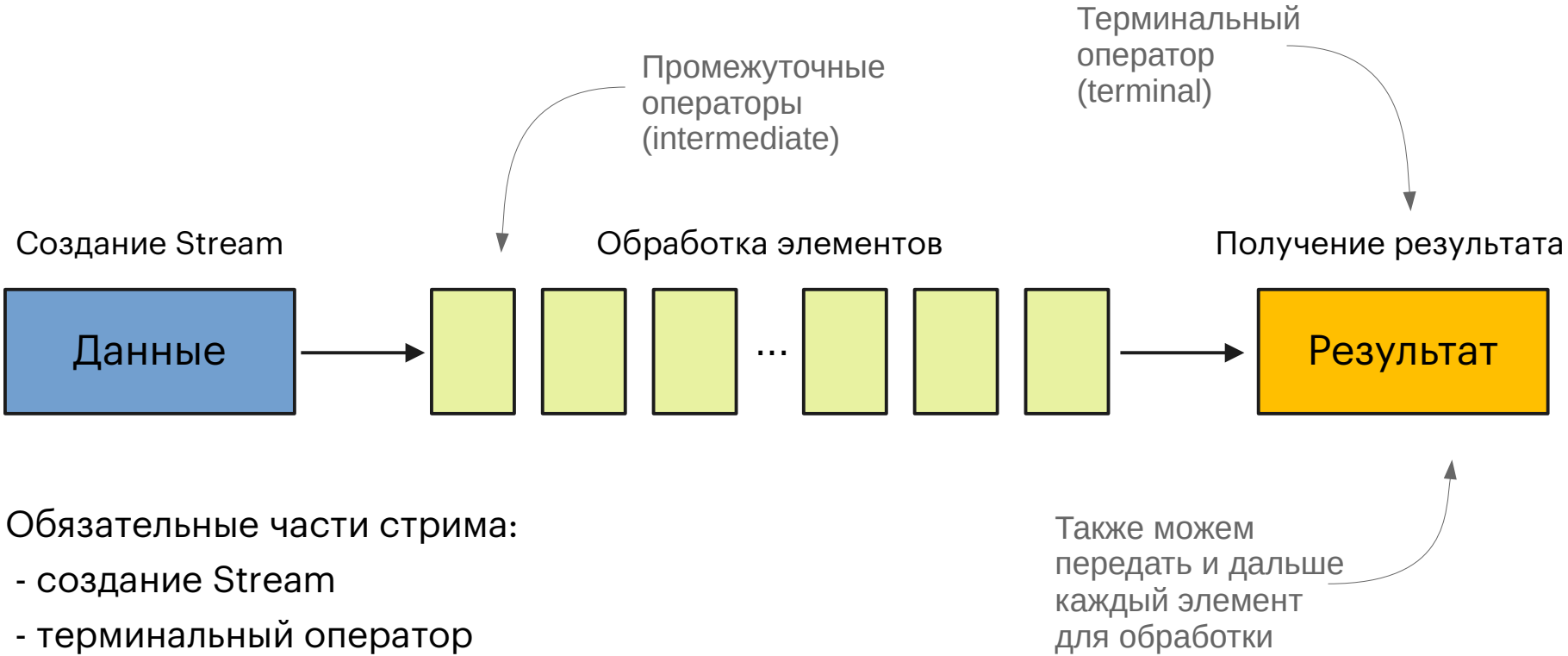
Stream (англ. поток) – объект для универсальной работы с наборами данных.

Появился в Java 8. Имеет типизацию <>

Визуально можно представить как конвейер, в который мы поставляем данные, к ним применяются фильтры, модификации, преобразования и результат работы пакуются в нужный формат или обрабатываются внешними методами.



Stream API



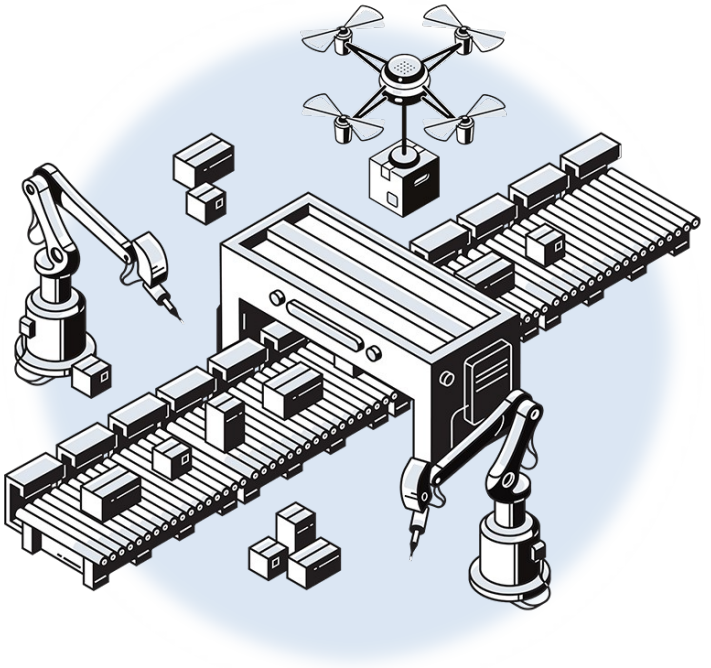
Обязательные части стрима:

- создание Stream
- терминальный оператор

Обработка добавляется по вкусу.

Stream API

создание



Skillbox

Stream API - создание

Пустой Stream – принимает тип в <>

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

Stream на основе перечисления

```
Stream<String> stream = Stream.of("a", "b", "c");
```

Stream на основе Collection (List, Set, Queue...), методы получения стрима stream()

```
List<Double> doubleList = List.of(1D, 2D, 3D);
Stream<Double> stream = doubleList.stream();
```

Stream на основе Map, получем EntrySet<> → это коллекция Set (по сути метод выше)

```
Map<String, Integer> map = Map.of("se", 2, "ddl", 3);
Stream<Entry<String, Integer>> stream = map.entrySet().stream();
```

Stream API - создание

Для создания Stream таким образом можно использовать массивы long, double, int.
В итоге получаются стримы LongStream, DoubleStream, IntStream.

```
double[] dArray = new double[]{3.4, 5.6};
DoubleStream stream = Arrays.stream(dArray);
```

Генерация элементов стрима, таким образом получается “бесконечный стрим”

```
Stream<Double> stream = Stream.generate(() -> Math.random());
```

лямбда

Генерация с ограничением количества элементов

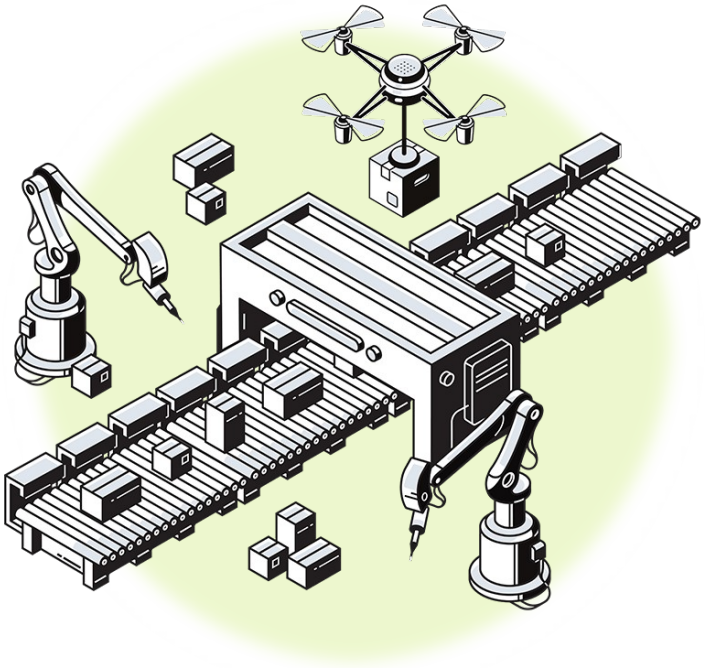
```
Stream<Double> stream = Stream.generate(Math::random).limit(10);
```

указатель на метод, короткая запись лямбда выражения

Это не все способы создания, остальные порождающие методы можно посмотрет в `java.util.stream.Stream` и наследниках.

Stream API

промежуточные операторы



Skillbox

Stream API – один терминальный метод для наглядности

В примерах, для визуальной оценки результатов, будем использовать терминальный оператор forEach(), его сигнатура:

```
void forEach(Consumer<? super T> action);
```

- метод void, то есть ничего не возвращает, на этом жизненный цикл стрима заканчивается.
- в аргументы принимает **Consumer** (Потребитель). Это функциональный интерфейс, что нам надо знать о нем – мы можем использовать методы, которые принимают один аргумент типа T (тип нашего стрима) и ничего не возвращают (void).
- нам подойдет такой метод как **System.out.println()** - принимает в аргументы любой объект и ничего не возвращает. Метод будет применен к каждому элементу стрима

```
Stream<String> stream = Stream.of("a", "bb", "ccc");
stream.forEach(str -> System.out.println(str));
```

намерено в примерах показаны лямбда выражения, не указатели на методы.

Skillbox

Stream API – промежуточные операторы

.map()

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

Преобразует элемент стрима типа T, а элемент типа R.
В итоге получим стрим типизированный по R.

```
Stream<String> stream = Stream.of("aa", "b", "cccc");
Stream<Integer> streamInt = stream.map(str -> str.length());
```

Происходит преобразование каждого элемента String в элемент Integer.
Используется метод length() который возвращает значение длины строки в int.
Неявно происходит упаковка в Integer. Далее, напечатаем каждый элемент стрима.

```
streamInt.forEach(x -> System.out.println(x));
```

Stream API – промежуточные операторы

.map()

Необязательно использовать метод, можно описать преобразование в виде лямбда строки:

```
Stream<Integer> stream = Stream.of(1, 2, 3);
Stream<Integer> streamInt = stream.map(i -> i * 2);
streamInt.forEach(i -> System.out.println(i));
```

```
2
4
6
```

Чаще это записывают в виде цепочки вызовов “одной строкой”:

```
Stream.of(1, 2, 3)
    .map(i -> i * 2)
    .forEach(System.out::println);
```

```
2
4
6
```

Так можно записать, если в методе только один аргумент, автоматически будет использовать элемент стрима.
:: разделяет класс и вызывыемый метод

Skillbox

Stream API – промежуточные операторы

.filter()

```
Stream<T> filter(Predicate<? super T> predicate);
```

Проверяет каждый элемент стрима по условию, **Predicate** возвращает **boolean**, а сам **filter()** возвращает stream того же типа и в него попадают только те элементы, которые при применении **predicate** вернули **true**.

```
Stream.of(1, 2, 3, 4, 5, 6)
    .filter(number -> number % 3 == 0)
    .forEach(System.out::println);
```

```
3
6
```

```
Stream.of("ski", "", "ll", "   ", "box")
    .filter(s -> !s.isBlank())
    .forEach(System.out::print);
```

```
skillbox
```

Skillbox

Stream API – промежуточные операторы

.filter()

Инвертировать проверку можно статическим методом **Predicate.not()**

```
Stream.of("ski", "", "ll", "   ", "box")
    .filter(Predicate.not(s -> s.isBlank()))
    .forEach(System.out::print);
```

И также использовать указатель на метод — красота:

```
Stream.of("ski", "", "ll", "   ", "box")
    .filter(Predicate.not(String::isBlank))
    .forEach(System.out::print);
```

Статический метод **isEqual()** сравнивает равенство объектов:

```
Stream.of("ski", "", "ll", "   ", "box")
    .filter(Predicate.isEqual("ski"))
    .forEach(System.out::print);
```

Stream API – промежуточные операторы

.distinct()

Фильтр, пропускает в новый стрим только уникальные элементы, проверка происходит методом **hashCode()** и **equals()**

```
Stream.of("skillbox", "sendel", "skillbox", "sendel")
    .distinct()
    .forEach(System.out::println);
```

```
skillbox
sendel
```

Проверка уникальности аналогична вставке в HashSet/HashMap — важен hashCode

Промежуточные операторы комбинируются, так как каждый возвращает Stream

```
Random random = new Random(0xCAFE); // рандом с указанием seed
Stream.generate(() -> random.nextInt(6)) //случайный int от 0 до 5 (вкл)
    .limit(10) // генерация 3 элементов в стрим
    .distinct() // удаляем повторы
    .map(x -> ++x) // добавляем +1 к каждому значению
    .forEach(System.out::println); // печать значений
```

```
2
1
5
4
6
```

Stream API – промежуточные операторы

.sorted()

Сортирует элементы стрима, если классы стрима имплементировали Comparable — аргумент не требуется

```
Stream.of("skillbox", "java", "art")
    .sorted()
    .forEach(System.out::println);
```

```
art
java
skillbox
```

иначе требуется передать компаратор (без него будет RuntimeException)

```
Stream.of(List.of(1,4,5), List.of(1), List.of(4, 7))
    .sorted(Comparator.comparing(List::size))
    .forEach(System.out::println);
```

```
[1]
[4, 7]
[1, 4, 5]
```

Stream API – промежуточные операторы

.limit()

Ограничивает количество элементов в стриме
начиная с первого до значения лимита.

```
Stream.of("skillbox", "java", "art", "linux")
    .sorted()
    .limit(2)
    .forEach(System.out::println);
```

```
art
java
```

ограничиваем количество элементов при генерации:

```
Random random = new Random(0xDECAF); // рандом с указанием seed
Stream.generate(random::nextInt) //случайный int
    .limit(5) // генерация 10 элементов в стрим
    .forEach(System.out::println); // печать значений
```

```
491928659
462567846
1918819131
169200396
1902109967
```

Skillbox

Stream API – промежуточные операторы

.peek()

применение метода к стриму, используется для логирования.

```
"abcd".chars() // получение IntStream из строки
    .peek(c -> System.out.println("char:" + (char) c))
    .forEach(System.out::println);
```

```
char:a
97
char:b
98
char:c
99
char:d
100
```

Обратите внимание на порядок, обработка идет одного элемента по всем операторам и после переход к следующему элементу.

.skip()

пропуск указанное количество первых элементов

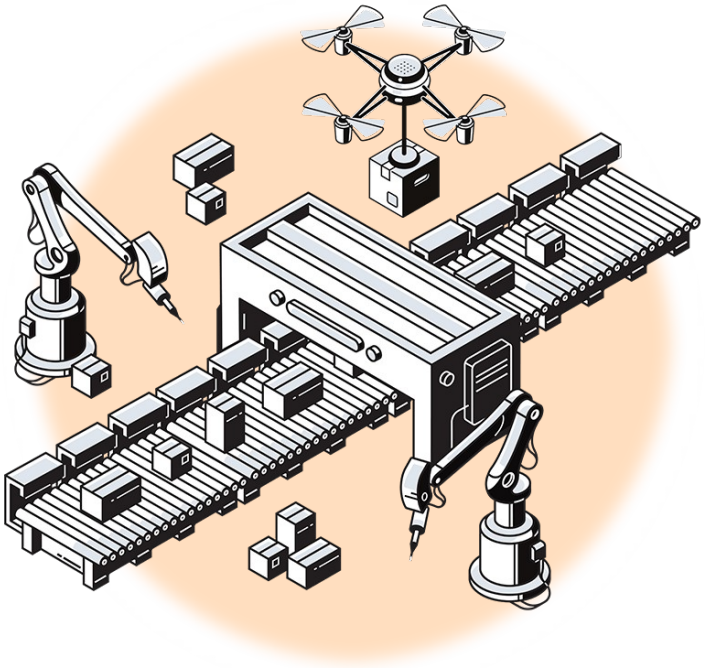
```
"abcd".chars() // получение IntStream из строки
    .peek(c -> System.out.println("char:" + (char)c))
    .skip(2)
    .forEach(System.out::println);
```

```
char:a
char:b
char:c
99
char:d
100
```

Элемент отбрасывается на этапе skip(), до этого все действия будут выполнены для каждого элемента по очереди.

Stream API

терминальные операторы



Skillbox

Stream API – терминальные операторы

.forEach() применение функции Consumer к каждому элементу стрима,

```
Stream.of("skillbox", "sendel", "skillbox", "sendel")
      .distinct()
      .forEach(x -> System.out.println("->" + x));
```

.count() возвращает long количества элементов стрима.

```
long count = Stream.of("skillbox", "sendel", "skillbox", "sendel")
     .distinct()
     .count(); //2
```

Stream API – терминальные операторы

.min() получает минимальный элемент стрима,
требует передачу Comparator

```
Optional<String> shortestString =Stream.of("a", "bb", "ccc", "d")
     .min(Comparator.comparing(String::length)); //Optional[a]
```

Получаем первый элемент из минимальных между a и d

.max() получает максимальный элемент стрима,
требует передачу Comparator

```
Optional<String> longestString =Stream.of("a", "bb", "ccc", "d")
     .max(Comparator.comparing(String::length)); //Optional[ccc]
```

про Optional<>

Skillbox

Stream API – Optional<>

В некоторых случаях мы требуем Stream вернуть значение или объект,
но объекта просто может на быть, например на примере max:

```
Optional<Integer> max = Stream.of(1, 2, 3, 4)
    .filter(i -> i > 10)
    .max(Integer::compare); //используем метод у Integer
```

после фильтра, у нас нет ни одного элемента с стриме, а значит нет ответа
на запрос максимального числа. И результат выполнения в таком случае
пакуется в специальный класс Optional, который хранит результат.

Результата может быть два:
- полученное значение, в нашем случае будет найденное максимальное значение
- empty, максимальное значение не найдено, пустой контейнер

Stream API – Optional<>

Таким образом решать, решение «что делать» если найдено или нет значение мы можем на основе вызова методов Optional:

```
 Optional<Integer> shortestString = Stream.of(1, 2, 3, 4)
     .filter(i -> i > 10)
     .max(Integer::compare);

if (shortestString.isPresent()){
    System.out.println(shortestString.get());
} else {
    System.out.println("Максимальное значение не найдено!");
}
```

Для проверки есть методы isEmpty() и isPresent(), для получения значения метод get().

Stream API – Optional<>

Можем сразу на этапе получения результата стрима решить:

- вернуть значение по умолчанию

```
int max = Stream.of(1, 2, 3, 4)
    .filter(i -> i > 10)
    .max(Integer::compare)
    .orElse(0)
```

- выбросить исключение

```
int max = Stream.of(1, 2, 3, 4)
    .filter(i -> i > 10)
    .max(Integer::compare)
    .orElseThrow(() -> new NotFoundMaxException());
```

Подробнее: Optional: Кот Шрёдингера в Java 8

Skillbox

Stream API – терминальные операторы

.anyMatch()

возвращает true, если найден хотя бы один элемент **соответствующий** условию

```
boolean hasMoreThan10 = Stream.of(1, 2, 3, 4)
    .anyMatch(i -> i > 10); //false
```

.noneMatch()

возвращает true, если все элементы **НЕ соответствуют** условию

```
boolean allMoreThan10 = Stream.of(1, 2, 3, 4)
    .noneMatch(i -> i > 10); //true
```

.allMatch()

возвращает true, если все элементы **соответствуют** условию

```
boolean allLessThan10 = Stream.of(1, 2, 3, 4)
    .allMatch(i -> i > 10); //false
```

Stream API – терминальные операторы

.findFirst()

возвращает первый элемент стрима

```
Optional<Integer> first = Stream.of(1, 20, 30, 40)
    .filter(i -> i > 10)
    .findFirst(); //20
```

.findAny()

возвращает любой элемент стрима

```
Optional<Integer> any = Stream.of(1, 20, 30, 40)
    .filter(i -> i > 10)
    .findAny(); //20
```

Разница: при многопоточном (параллельном) стриме, findAny() необязательно вернет первый по порядку, а первый который был обработан в одном из потоков, порядок возврата не гарантируется при findAny()

```
Optional<Integer> any = Stream.of(10, 20, 30, 100)
    .parallel()
    .findAny(); // 10 или 20 или 30 или 100
```

```
Optional<Integer> any = Stream.of(10, 20, 30, 100)
    .parallel()
    .findFirst(); // 10
```

Stream API – терминальные операторы

.collect()

Собирает стрим в нужную структуру данных или объект

Один вариант принимает класс Collector, который уже содержим обработки

```
<R, A> R collect(Collector<? super T, A, R> collector);
```

Перегруженный метод возможно использовать для написания своей обработки

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

Stream API – терминальные операторы

Существующие методы получения классов Collector в JDK находятся в `java.util.stream.Collectors`

.toCollection() с выбором конструктора коллекции

```
Collection<Integer> collection = Stream.of(10, 20, 30, 100)
    .collect(Collectors.toCollection(LinkedList::new));
```

```
Set<Integer> collection = Stream.of(10, 100, 5, 100)
    .collect(Collectors.toCollection(TreeSet::new));
```

```
Queue<Integer> collection = Stream.of(10, 100, 5, 100)
    .collect(Collectors.toCollection(ConcurrentLinkedDeque::new));
```

Возвращает коллекция именно того типа что и передали:

```
ArrayList<Integer> collection = Stream.of(10, 100, 5, 100)
    .collect(Collectors.toCollection(ArrayList::new));
```

Stream API – терминальные операторы

.toList() короткая запись получения List (реализация ArrayList)

```
List<Integer> collection = Stream.of(10, 100, 5, 100)
    .collect(Collectors.toList());
```

.toSet() короткая запись получения Set (реализация HashSet)

```
Set<Integer> collection = Stream.of(10, 100, 5, 100)
    .collect(Collectors.toSet());
```

.toUnmodifiableList() неизменяемый List (реализация ArrayList)

JDK 10

```
List<Integer> collection = Stream.of(10, 100, 5, 100)
    .collect(Collectors.toUnmodifiableList());
```

.toUnmodifiableSet() неизменяемый List (реализация HashSet)

JDK 10

```
Set<Integer> collection = Stream.of(10, 100, 5, 100)
    .collect(Collectors.toUnmodifiableSet());
```

При попытке изменить коллекцию, будет выброшено исключение `java.lang.UnsupportedOperationException`

Skillbox

Stream API – терминальные операторы

.toMap() получения различными способами Map

Для примера будем работать с классом Person

```
public static class Person {

    private final String phone;

    public Person(String phone) {
        this.phone = phone;
    }

    public String getPhone() {
        return phone;
    }

    @Override
    public String toString() {
        return "Person{" + "phone='" + phone + '\'' + '}';
    }
}
```

Skillbox

Stream API – терминальные операторы

.toMap()

Пример сборки Map, где ключ это телефон, а значение это сам Person

```
Map<String, Person> map =
    Stream.of(new Person("1"), new Person("22"), new Person("3"))
        .collect(Collectors.toMap(Person::getPhone, Function.identity()));
```

Вставляет в **Key** полученное значение из элемента стрима

Вставляет в **Value** сам элемент стрима

Результат:

```
{22=Person{phone='22'}, 1=Person{phone='1'}, 3=Person{phone='3'}}
```

Stream API – терминальные операторы

.toMap()

Если ключи будут одинаковы → выбрасывается java.lang.IllegalStateException

```
Map<String, Person> map =
    Stream.of(new Person("1"), new Person("1"))
        .collect(Collectors.toMap(Person::getPhone, Function.identity()));
```

Результат:

```
Exception in thread "main" java.lang.IllegalStateException Create breakpoint : Duplicate key 1 (attempted merging values Person{phone='1'} and Person{phone='1'})
  at java.base/java.util.stream.Collectors.duplicateKeyException(Collectors.java:135)
  at java.base/java.util.stream.Collectors.lambda$uniqKeysMapAccumulator$1(Collectors.java:182) <6 internal lines>
  at java.base/java.util.stream.ReferencePipeline.collect(ReferencePipeline.java:682)
  at scratch.main(scratch.java:30)
```

Решение — указать явно что делать при дубликате:

```
Map<String, Person> map =
    Stream.of(new Person("1"), new Person("1"))
        .collect(Collectors.toMap(
            Person::getPhone, Function.identity(),
            (existing, current) -> existing));
```

третий аргумент это **BinaryOperator<U> mergeFunction**, то есть принимает два аргумента и возвращает один. В нашем случае явно указываем, что записать в **Value** к существующему **Key**

Stream API – терминальные операторы

.toMap() подсчет количества элементов на основе предыдущего кода

```
Map<String, Integer> map =
    Stream.of("a", "a", "c", "b", "c", "a")
        .collect(Collectors.toMap(
            Function.identity(),
            (s)-> 1,
            (existing, current) -> ++existing));
```

Результат:

```
{a=3, b=1, c=2}
```

Stream API – терминальные операторы

.groupBy() группировка данных, для получения Map

Подсчет количества элементов через группировку данных

```
Map<String, Long> map =
    Stream.of("a", "a", "c", "b", "c", "a")
        .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
```

Результат:

```
{a=3, b=1, c=2}
```

Собрать элементы с одинаковым признаком в List:

```
Map<Integer, List<Person>> map =
    Stream.of(new Person("2"), new Person("3"), new Person("33"), new Person("55"))
        .collect(Collectors.groupingBy((p) -> p.getPhone().length()));
```

Результат:

```
{1=[Person{phone='2'}, Person{phone='3'}], 2=[Person{phone='33'}, Person{phone='55'}]}
```

https://habr.com/ru/post/348536/

Stream API – терминальные операторы

.groupBy() группировка данных, для получения Map

Собрать элементы с одинаковым признаком в Set:

```
Map<Integer, Set<Person>> map =
    Stream.of(new Person("2"), new Person("3"), new Person("33"), new Person("55"))
        .collect(Collectors.groupingBy((p)->p.getPhone().length(), Collectors.toSet()));
```

https://habr.com/ru/post/348536/

Stream API – терминальные операторы

.groupBy()

Для примера будем работать с классом Product

```
public class Product {

    private final String type;
    private final long price;

    public Product(String type, long price) {
        this.type = type;
        this.price = price;
    }

    public String getType() { return type; }

    public long getPrice() { return price; }
}
```

Stream API – терминальные операторы

.groupBy()

Посчитать сумму стоимости товаров в одной категории:

```
Map<String, Long> map =
    Stream.of(
            new Product("milk", 60),
            new Product("bread", 30),
            new Product("milk", 40),
            new Product("bread", 200))
        .collect(Collectors.groupingBy(
            Product::getType,
            Collectors.summingLong(Product::getPrice)));
```

Результат:

```
{bread=230, milk=100}
```

https://habr.com/ru/post/348536/

Ссылки и источники

- Шпаргалка Java программиста 4. Java Stream API
  https://habr.com/ru/company/luxoft/blog/270383/
- Java 8 Collectors toMap
  https://www.baeldung.com/java-collectors-tomap
- 7 способов использовать groupingBy в Stream API
  https://habr.com/ru/post/348536/
- Функциональные интерфейсы и лямбда-выражения в Java
  https://skillbox.ru/media/base/funktsionalnye_interfeysy_i_lyambda_vyrazheniya_v_java/

Бонус

ответы на вопросы участников

Skillbox

1. Как производится обработка элементов в стриме? Каждый проходит всю цепочку методов и после выполняения терминального оператора приступаем к обработке следующего элемента?

Предлагаю такой код для проверки этого вопроса:

```
public static void main(String[] args) {
    long startTime = System.currentTimeMillis();
   Stream.of(1,2,3)
       .peek(Main::sleep)
       .map(integer -> integer *2)
       .peek(Main::sleep)
       .forEach(integer -> System.out.printf(
               "Элемент %d, время от старта: %d%n",
           integer,System.currentTimeMillis() - startTime));
}

public static void sleep(int i) {
    try { Thread.sleep(1000); }
    catch (InterruptedException e) { e.printStackTrace(); }
}
```

```
Элемент 2, время от старта: 2010
Элемент 4, время от старта: 4013
Элемент 6, время от старта: 6014
```

В каждой цепочке методов используется два вызова **sleep()**. Наглядно нам показывает, в обычном стриме просходит работа с каждым элементов по очереди, как закончили обработку одного, переходим к следующему.

1. Как производится обработка элементов в стриме? Каждый проходит всю цепочку методов и после выполняения терминального оператора приступаем к обработке следующего элемента?

Если использовать паралельный стрим:

```
public static void main(String[] args) {
    long startTime = System.currentTimeMillis();
   Stream.of(1,2,3)
       .parallel()
       .peek(Main::sleep)
       .map(integer -> integer *2)
       .peek(Main::sleep)
       .forEach(integer -> System.out.printf(
               "Элемент %d, время от старта: %d%n",
           integer,System.currentTimeMillis() - startTime));
}

public static void sleep(int i) {
    try { Thread.sleep(1000); }
    catch (InterruptedException e) { e.printStackTrace(); }
}
```

```
Элемент 4, время от старта: 2012
Элемент 6, время от старта: 2013
Элемент 2, время от старта: 2013
```

Каждый элемент обрабатывается в своем потоке и каждый проходит свою цепочку параллельно другим, поэтому закончат все в одно время. При этом порядок вывода не гарантируется.

Гарантирует сохранность порядка:

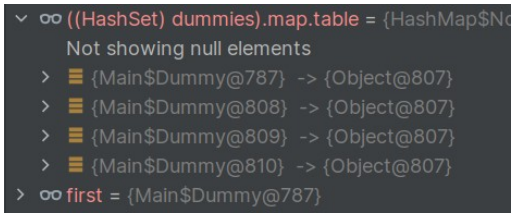
`.forEachOrdered()`

2. Какой элемент будет возвращен, если запросить у Stream findFirst(), полученный из HashSet?

Для этого создададим HashSet, на основе класса пустышки, и посмотрим расположение элементов в массиве:

```
public static void main(String[] args) {
Set<Dummy> dummies =
    new HashSet<>(List.of(new Dummy(), new Dummy(),
            new Dummy(),new Dummy()));

Dummy first = dummies.stream()
    .findFirst().orElseThrow();
}

public static class Dummy{}
```

В дебаге смотрим, какой элемент в first (несколько запусков):



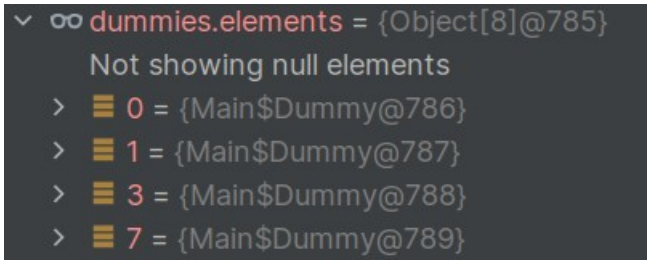
Если менять формирование hashCode у Dummy, по разному заполнять HashSet, то в проведенных запусках, всегда возвращается первый элемента массива table.

Но не всегда это так, если мы получаем объект типа Set, то мы не можем знать точную реализацию, а от этого зависит работа findFirst(), смотрите следующий слайд.

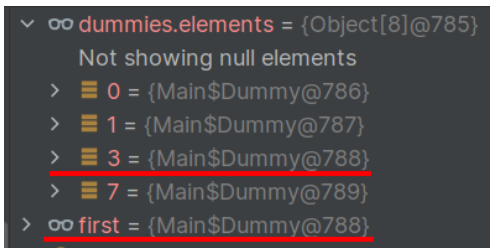
2. Какой элемент будет возвращен, если запросить у Stream findFirst(), полученный из HashSet?

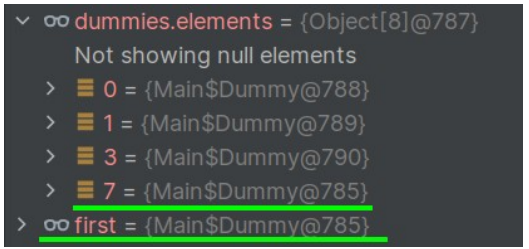
А теперь у нас будет `java.util.ImmutableCollections$SetN` (получение Set через Set.of()), и тут findFirst() ведет себя по другому:

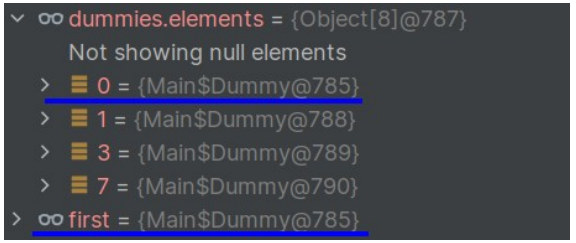
```
public static void main(String[] args) {
    Set<Dummy> dummies =
        Set.of(new Dummy(), new Dummy(),
               new Dummy(),new Dummy());

   Dummy first = dummies.stream()
       .findFirst().orElseThrow();
}

public static class Dummy{}
```



В дебаге смотрим, какой элемент в first (несколько запусков):







Можно сделать вывод, что заранее спрогнозировать полученный элемент на основе порядкого номера массива elements не получится.

Skillbox

3. Как слить несколько стримов через flatMap?

Если стримы пришли в коллекции:

```
Collection<Stream<String>> streamList =
    List.of(
        Stream.of("1", "10", "100"),
        Stream.of("2", "20", "200"));

Stream<String> stream = streamList.stream()
    .flatMap(stringStream -> stringStream);

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

Во flatMap необходимо передать функцию, которая из элементов текущего стрима, получит новый стрим. Если у нас есть список стримов, достатачно вызывать, как на верхнем коде.

Или сформировать стримы из элементов, как показано справа.

Если формируем стримы из элементов коллекции

```
Collection<String> strings =
    List.of("1 10 100", "2 20 200", "3 30 300");

Stream<String> stream = strings.stream()
    .flatMap(str -> Arrays.stream(str.split("\\s+")));

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

или заранее переведем в массив, а после получим Stream:

```
Collection<String> strings =
    List.of("1 10 100", "2 20 200", "3 30 300");

Stream<String> stream = strings.stream()
    .map(str -> str.split("\\s+"))
    .flatMap(Arrays::stream);

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

Вебинар

Спасибо за внимание!

Предложения и вопросы:
tg: @sendel
sendel@sendel.ru

Спикер: Шибков Константин