STREAMS – СТРИМЫ

**Stream API** это новый (с Java8) способ работать со структурами данных в функциональном стиле. Это поток данных коллекции или массива.

**Способы создания Stream:**

1. *Классический: Создание стрима из коллекции*

Collection<String> collection = Arrays.asList("a1", "a2", "a3");

Stream<String> streamFromCollection = collection.stream();

*// Создание стрима из коллекции*

Collection<String> collection = Arrays.asList("a1", "a2", "a3");

Stream<String> streamFromCollection = collection.stream();

System.out.println("streamFromCollection = " + streamFromCollection.collect(Collectors.toList()));

*// напечатает streamFromCollection = [a1, a2, a3]*

1. *Создание стрима из значений*

Stream<String> streamFromValues = Stream.of("a1", "a2", "a3");

*// Создание стрима из значений*

Stream<String> streamFromValues = Stream.of("a1", "a2", "a3");

System.out.println("streamFromValues = " + streamFromValues.collect(Collectors.toList()));

*// напечатает streamFromValues = [a1, a2, a3]*

1. *Создание стрима из массива*

String[] array = {"a1","a2","a3"};

Stream<String> streamFromArrays = Arrays.stream(array);

*// Создание стрима из массива*

String[] array = {"a1","a2","a3"};

Stream<String> streamFromArrays = Arrays.stream(array);

System.out.println("streamFromArrays = " + streamFromArrays.collect(Collectors.toList()));

*// напечатает streamFromArrays = [a1, a2, a3]*

или

Stream<String> streamFromArrays1 = Stream.of(array);

System.out.println("streamFromArrays1 = " + streamFromArrays1.collect(Collectors.toList()));

*// напечатает streamFromArrays = [a1, a2, a3]*

1. *Создание стрима из файла (каждая строка в файле будет отдельным элементом в стриме)*

Stream<String> streamFromFiles = Files.lines(Paths.get("file.txt"));

*// Создание стрима из файла (каждая запись в файле будет отдельной строкой в стриме)*

File file = new File("1.tmp");

file.deleteOnExit();

PrintWriter out = new PrintWriter(file);

out.println("a1");

out.println("a2");

out.println("a3");

out.close();

Stream<String> streamFromFiles = Files.lines(Paths.get(file.getAbsolutePath())); System.out.println("streamFromFiles = " + streamFromFiles.collect(Collectors.toList())); *// напечатает streamFromFiles = [a1, a2, a3]*

1. *Создание стрима из строки*

IntStream streamFromString = "123".chars();

*// Создание числового стрима из строки*

IntStream streamFromString = "123".chars();

System.out.print("streamFromString = ");

streamFromString.forEach((e)->System.out.print(e + " , "));

*// напечатает streamFromString = 49 , 50 , 51 ,* - соответствует

1. *С помощью Stream.builder*

Stream<Object> stream = Stream.builder().add("a1").add("a2").add("a3").build();

*// С помощью Stream.builder*

Stream.Builder<String> builder = Stream.builder();

Stream<String> streamFromBuilder = builder.add("a1").add("a2").add("a3").build(); System.out.println("streamFromBuilder = " + streamFromBuilder.collect((Collectors.toList())));

*// напечатает streamFromFiles = [a1, a2, a3]*

1. *Создание бесконечного стрима с помощью Stream.iterate* (*начальное условие*, *выражение генерации*)

Stream<Integer> streamFromIterate = Stream.iterate(1, n -> n + 1);

*// С помощью Stream.iterate*

Stream<Integer> streamFromIterate = Stream.iterate(1, n -> n + 2);

System.out.println("streamFromIterate = " + streamFromIterate.limit(3).collect(Collectors.toList()));

*// напечатает streamFromIterate = [1, 3, 5]*

1. *Создание бесконечного стрима с помощью Stream.generate (выражение генерации)*

Stream<String> streamFromGenerate = Stream.generate(() -> "a1");

*// С помощью Stream.generate*

Stream<String> streamFromGenerate = Stream.generate(() -> "a1"); System.out.println("streamFromGenerate = " + streamFromGenerate.limit(3).collect(Collectors.toList()));

*// напечатает streamFromGenerate = [a1, a1, a1]*

1. *Создание параллельного стрима*

Stream<String> stream = collection.parallelStream();

*// Создать параллельный стрим из коллекции*

Stream<String> parallelStream = collection.parallelStream();

System.out.println("parallelStream = " + parallelStream.collect(Collectors.toList()));

*// напечатает parallelStream = [a1, a2, a3]*

*+10 - // Создать пустой стрим*

Stream<String> streamEmpty = Stream.empty();

System.out.println("streamEmpty = " + streamEmpty.collect(Collectors.toList()));

*// напечатает streamEmpty = []*

**Методы (операции) STREAM:**

* -Intermediate - lazy (конвейерные)
* -Terminal - eager (терминальные)

У stream'a может быть сколько угодно вызовов конвейерных вызовов и в конце один терминальный, при этом все конвейерные методы выполняются лениво и пока не будет вызван терминальный метод никаких действий на самом деле не происходит.

Это как создать объект Thread или Runnable, но не вызвать у него start.

**Основные конвейерные:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Метод stream** | **Описание** | **Пример** | | **filter** | Отфильтровывает записи, возвращает только записи, соответствующие условию | collection.stream().filter(«a1»::equals).count() | | **skip** | Позволяет пропустить N первых элементов | collection.stream().skip(collection.size() — 1)  .findFirst().orElse(«1») | | **distinct** | Возвращает стрим без дубликатов (для метода equals)  Есть упорядоченный и неупорядоченный способ (см ниже) | collection.stream().distinct().collect(Collectors.toList()) | | **map** | Преобразует каждый элемент стрима | collection.stream().map((s) -> s + "\_1")  .collect(Collectors.toList()) | | **Peek (типа forEach только на ней не заканчивается)** | Возвращает тот же стрим, но применяет функцию к каждому элементу стрима | collection.stream().map(String::toUpperCase)  .peek((e) -> System.out.print("," + e)). collect(Collectors.toList()) | | **limit** | Позволяет ограничить выборку определенным количеством первых элементов | collection.stream().limit(2).collect(Collectors.toList()) | | **sorted** | Позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator | collection.stream().sorted().collect(Collectors.toList()) | | **mapToInt**, **mapToDouble**, **mapToLong** | Аналог map, но возвращает числовой стрим (то есть стрим из числовых примитивов) | collection.stream()  .mapToInt((s) -> Integer.parseInt(s)).toArray() | | **flatMap**, **flatMapToInt**, **flatMapToDouble**, **flatMapToLong** | Похоже на map, но может создавать из одного элемента несколько | collection.stream().flatMap((p) -> Arrays.asList(p.split(",")).stream()).toArray(String[]::new) | |

**Основные терминальные:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Метод stream** | **Описание** | **Пример** | | **findFirst** | Возвращает первый элемент из стрима (возвращает Optional) | collection.stream().findFirst().orElse(«1») | | **findAny** | Возвращает любой подходящий элемент из стрима (возвращает Optional) | collection.stream().findAny().orElse(«1») | | **collect** | Представление результатов в виде коллекций и других структур данных | collection.stream()  .filter((s) -> s.contains(«1»))  .collect(Collectors.toList()) | | **count** | Возвращает количество элементов в стриме | collection.stream().filter(«a1»::equals).count() | | **anyMatch** | Возвращает true, если условие выполняется хотя бы для одного элемента | collection.stream().anyMatch(«a1»::equals) | | **noneMatch** | Возвращает true, если условие не выполняется ни для одного элемента | collection.stream().noneMatch(«a8»::equals) | | **allMatch** | Возвращает true, если условие выполняется для всех элементов | collection.stream()  .allMatch((s) -> s.contains(«1»)) | | **min** | Возвращает минимальный элемент, в качестве условия использует компаратор | collection.stream()  .min(String::compareTo).get() | | **max** | Возвращает максимальный элемент, в качестве условия использует компаратор | collection.stream()  .max(String::compareTo).get() | | **forEach** | Применяет функцию к каждому объекту стрима, порядок при параллельном выполнении не гарантируется | set.stream().forEach((p) -> p.append("\_1")); | | **forEachOrdered** | Применяет функцию к каждому объекту стрима, сохранение порядка элементов гарантирует | list.stream()  .forEachOrdered((p) -> p.append("\_new")); | | **toArray** | Возвращает массив значений стрима | collection.stream()  .map(String::toUpperCase)  .toArray(String[]::new); | | **reduce** | Позволяет выполнять агрегатные функции на всей коллекцией и возвращать один результат | collection.stream()  .reduce((s1, s2) -> s1 + s2).orElse(0) | |

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **get** | Возвращает элемент  (один после фильтра или типа того)  или выводит ошибку | collection.stream()  .min(String::compareTo).get() | |

Методы findFirst, findAny, anyMatch это short-circuiting методы, то есть обход стримов организуется таким образом чтобы найти подходящий элемент максимально быстро, а не обходить весь изначальный стрим – ищет первый.

**+ дополнительные методы у числовых стримов**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Метод stream** | **Описание** | **Пример** | | **sum** | Возвращает сумму всех чисел | collection.stream()  .mapToInt((s) -> Integer.parseInt(s)).sum() | | **average** | Возвращает среднее арифметическое всех чисел | collection.stream()  .mapToInt((s) -> Integer.parseInt(s)).average() | | **mapToObj** | Преобразует числовой стрим обратно в объектный | intStream.mapToObj((id) -> new Key(id)).toArray() | |

**+ дополнительные методы для параллельных стримов:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **isParallel** | Узнать является ли стрим параллельным | | **parallel** | Вернуть параллельный стрим, если стрим уже параллельный,  то может вернуть самого себя | | **sequential** | Вернуть последовательный стрим, если стрим уже последовательный,  то может вернуть самого себя | |

С помощью, методов parallel и sequential можно определять какие операции могут быть параллельными, а какие только последовательными. Так же из любого последовательного стрима можно сделать параллельный и наоборот, то есть:

collection.stream(). peek(...). *// операция последовательна*

parallel(). map(...). *// операция может выполняться параллельно,* sequential(). reduce(...) *// операция снова последовательна*

**Внимание**: крайне не рекомендуется использовать параллельные стримы для сколько-нибудь долгих операций (получение данных из базы, сетевых соединений), так как все параллельные стримы работают c одним пулом fork/join и такие долгие операции могут остановить работу всех параллельных стримов в JVM из-за того отсутствия доступных потоков в пуле, т.е. параллельные стримы стоит использовать лишь для коротких операций, где счет идет на миллисекунды, но не для тех где счет может идти на секунды и минуты.

**Примеры применения методов STRAEM’ов**

### Примеры использования filter, findFirst, findAny, skip, limit и count

### 1. Не Car()

public class TestStreamEx {  
  
 public static void main(String[] args) {  
  
 List<String> collection = Arrays.*asList*("a1", "a2", "a3", "a1");  
  
*// Вернуть количество вхождений объекта «a1»* System.*out*.println(collection.stream().filter("a1"::equals).count());  
*//2  
  
// Вернуть первый элемент коллекции или 0, если коллекция пуста* System.*out*.println(collection.stream().findFirst().orElse("0"));  
*// a1  
  
// Вернуть последний элемент коллекции или «empty», если коллекция пуста* System.*out*.println(collection.stream().skip(collection.size()-1).findAny().orElse("empty"));  
*// a1  
  
// Найти элемент в коллекции равный «a3» или кинуть ошибку* System.*out*.println(collection.stream().filter(x->x.equals("a35")).findAny().get());  
*// java.util.NoSuchElementException  
  
// Вернуть третий элемент коллекции по порядку* System.*out*.println(collection.stream().skip(2).findFirst().get());  
*// a3  
  
// если больше элементов чем надо, то orElse ек даст ошибку* System.*out*.println(collection.stream().skip(10).findFirst().orElse("Lox"));  
*// Lox  
  
// Вернуть два элемента начиная со второго* collection.stream().skip(1).limit(2).forEach(x-> System.*out*.print(x + " "));  
*// или* System.*out*.println(Arrays.*toString*(collection.stream().skip(1).limit(2).toArray()));  
*// a2 a3, [a2, a3]  
  
// Выбрать все элементы по шаблону* collection.stream().filter(x->x.contains("1")).forEach(x-> System.*out*.print(x + " "));  
*// a1 a1* }  
}

**Внимание:** findFirst и findAny возвращают новый тип Optional, появившийся в Java 8, для того чтобы избежать NullPointerException

### \_\_

### 2. CAR()

**Условие**: дана коллекция класс People (с полями name — имя, age — возраст, sex — пол), вида Arrays.asList( new People(«Вася», 16, Sex.MAN), new People(«Петя», 23, Sex.MAN), new People(«Елена», 42, Sex.WOMEN), new People(«Иван Иванович», 69, Sex.MAN)).

private static void testFilterAndCount() {

private enum Sex {

MAN, WOMEN }

private static class People {

private final String name;

private final Integer age;

private final Sex sex;

public People(String name, Integer age, Sex sex) {

this.name = name;

this.age = age;

this.sex = sex;

}

public String getName() {

return name;

}

public Integer getAge() {

return age;

}

public Sex getSex() {

return sex;

}

@Override public String toString() {

return "{" + "name='" + name + '\'' + ", age=" + age + ", sex=" + sex + '}';

}

psvm {

Collection<People> peoples = Arrays.asList( new People("Вася", 16, Sex.MAN), new People("Петя", 23, Sex.MAN), new People("Елена", 42, Sex.WOMEN), new People("Иван Иванович", 69, Sex.MAN) );

*// Выбрать мужчин-военнообязанных*

List<People> militaryService = peoples.stream().filter((p)-> p.getAge() >= 18 && p.getAge() < 27 && p.getSex() == Sex.MAN).collect(Collectors.toList());

System.out.println("militaryService = " + militaryService);

*// напечатает militaryService = [{name='Петя', age=23, sex=MAN}]* *//*

*Найти средний возраст среди мужчин*

double manAverageAge = peoples.stream().filter((p) -> p.getSex() == Sex.MAN). mapToInt(People::getAge).average().getAsDouble();

System.out.println("manAverageAge = " + manAverageAge);

*// напечатает manAverageAge = 36.0*

*// Найти кол-во потенциально работоспособных людей в выборке (т.е. от 18 лет и учитывая что женщины выходят в 55 лет, а мужчина в 60)*

long peopleHowCanWork = peoples.stream().filter((p) -> p.getAge() >= 18).filter( (p) -> (p.getSex() == Sex.WOMEN && p.getAge() < 55) || (p.getSex() == Sex.MAN && p.getAge() < 60)).count(); System.out.println("peopleHowCanWork = " + peopleHowCanWork); *// напечатает manAverageAge = 2* }}

### Примеры использования distinct

Метод distinct возвращает stream без дубликатов, при этом для упорядоченного стрима (например, коллекция на основе list) порядок стабилен, для неупорядоченного — порядок не гарантируется. Рассмотрим результаты работы над коллекцией Collection ordered = Arrays.asList(«a1», «a2», «a2», «a3», «a1», «a2», «a2») и Collection nonOrdered = new HashSet<>(ordered).

Collection<String> ordered = Arrays.asList("a1", "a2", "a2", "a3", "a1", "a2", "a2"); Collection<String> nonOrdered = new HashSet<>(ordered);

*// Получение коллекции без дубликатов*

List<String> distinct =

nonOrdered.stream().distinct().collect(Collectors.toList()); System.out.println("distinct = " + distinct);

*// напечатает distinct = [a1, a2, a3] - порядок не гарантируется*

List<String> distinctOrdered = ordered.stream().distinct().collect(Collectors.toList()); System.out.println("distinctOrdered = " + distinctOrdered); *// напечатает distinct = [a1, a2, a3] - порядок гарантируется*

**Обратите внимание:**  
1. Если вы используете distinct с классом, у которого переопределен equals, обязательно так же корректно переопределить hashCode в соответствие с контрактом equals/hashCode (самое главное чтобы hashCode для всех equals объектов, возвращал одинаковое значение), иначе distinct может не удалить дубликаты (аналогично, как при использовании HashSet/HashMap),  
2. Если вы используете параллельные стримы и вам не важен порядок элементов после удаления дубликатов — намного лучше для производительности сделать сначала стрим неупорядоченным с помощь unordered(), а уже потом применять distinct(), так как подержание стабильности сортировки при параллельном стриме довольно затратно по ресурсам и distinct() на упорядоченным стриме будет выполнятся значительно дольше чем при неупорядоченном,

**Примеры использования Match функций (anyMatch, allMatch, noneMatch)**

*// Метод anyMatch - возвращает true, если условие выполняется хотя бы для одного элемента* *// Метод noneMatch - возвращает true, если условие не выполняется ни для одного элемента* *// Метод allMatch - возвращает true, если условие выполняется для всех элементов*

private static void testMatch() {

Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");

*// найти существуют ли хоть одно совпадение с шаблоном в коллекции*

boolean isAnyOneTrue = collection.stream().anyMatch("a1"::equals); System.out.println("anyOneTrue " + isAnyOneTrue);

*// напечатает true*

boolean isAnyOneFalse = collection.stream().anyMatch("a8"::equals); System.out.println("anyOneFlase " + isAnyOneFalse);

*// напечатает false*

*// найти существуют ли все совпадения с шаблоном в коллекции*

boolean isAll = collection.stream().allMatch((s) -> s.contains("1")); System.out.println("isAll " + isAll);

*// напечатает false*

*// сравнение на неравенство* boolean isNotEquals = collection.stream().noneMatch("a7"::equals); System.out.println("isNotEquals " + isNotEquals);

*// напечатает true* }

### Примеры использования Map функций (map, mapToInt, FlatMap, FlatMapToInt)

**Условие**: даны две коллекции collection1 = Arrays.asList(«a1», «a2», «a3», «a1») и collection2 = Arrays.asList(«1,2,0», «4,5»)

Обратите внимание: все map функции могут вернуть объект другого типа (класса), то есть map может работать со стримом строк, а на выходе дать Stream из значений Integer или получать класс людей People, а возвращать класс Office, где эти люди работают и т.п., flatMap (flatMapToInt и т.п.) на выходе должны возвращать стрим с одним, несколькими или ни одним элементов для каждого элемента входящего стрима (см. последние два примера).

*// Метод Map изменяет выборку по определенному правилу, возвращает stream с новой выборкой*

private static void testMap() {

Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");

*// Изменение всех элементов коллекции*

List<String> transform = collection.stream().map((s) -> s + "\_1").collect(Collectors.toList()); System.out.println("transform = " + transform);

*// напечатает transform = [a1\_1, a2\_1, a3\_1, a1\_1]*

*// убрать первый символ и вернуть числа*

List<Integer> number = collection.stream().map((s) -> Integer.parseInt(s.substring(1))).collect(Collectors.toList()); System.out.println("number = " + number);

*// напечатает transform = [1, 2, 3, 1]* }

*// Метод MapToInt - изменяет выборку по определенному правилу, возвращает stream с новой числовой выборкой*

private static void testMapToInt() {

Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");

*// убрать первый символ и вернуть числа*

int[] number = collection.stream().mapToInt((s) -> Integer.parseInt(s.substring(1))).toArray();

System.out.println("number = " + Arrays.toString(number));

*// напечатает number = [1, 2, 3, 1]* }

*// Метод FlatMap - похоже на Map - только вместо одного значения, он возвращает целый stream значений*

private static void testFlatMap() {

Collection<String> collection = Arrays.asList("1,2,0", "4,5");

*// получить все числовые значения, которые хранятся через запятую в collection*

String[] number = collection.stream().flatMap((p) -> Arrays.asList(p.split(",")).stream()).toArray(String[]::new); System.out.println("number = " + Arrays.toString(number));

*// напечатает number = [1, 2, 0, 4, 5]* }

*// Метод FlatMapToInt - похоже на MapToInt - только вместо одного значения, он возвращает целый stream значений*

private static void testFlatMapToInt() {

Collection<String> collection = Arrays.asList("1,2,0", "4,5");

*// получить сумму всех числовые значения, которые хранятся через запятую в collection*

int sum = collection.stream().flatMapToInt((p) -> Arrays.asList(p.split(",")).stream().mapToInt(Integer::parseInt)).sum(); System.out.println("sum = " + sum);

*// напечатает sum = 12* }

### Примеры использования Sorted функции (использует CompareTo)

**Условие**: даны две коллекции коллекция строк Arrays.asList(«a1», «a4», «a3», «a2», «a1», «a4») и коллекция людей класса People (с полями name — имя, age — возраст, sex — пол), вида Arrays.asList( new People(«Вася», 16, Sex.MAN), new People(«Петя», 23, Sex.MAN), new People(«Елена», 42, Sex.WOMEN), new People(«Иван Иванович», 69, Sex.MAN))

*// Метод Sorted позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator*

private static void testSorted() {

*// \*\*\*\*\*\*\*\*\*\*\*\* Работа со строками*

Collection<String> collection = Arrays.asList("a1", "a4", "a3", "a2", "a1", "a4");

*// отсортировать значения по алфавиту*

List<String> sorted = collection.stream().sorted().collect(Collectors.toList()); System.out.println("sorted = " + sorted);

*// напечатает sorted = [a1, a1, a2, a3, a4, a4]*

*// отсортировать значения по алфавиту и убрать дубликаты*

List<String> sortedDistinct = collection.stream().sorted().distinct().collect(Collectors.toList()); System.out.println("sortedDistinct = " + sortedDistinct);

*// напечатает sortedDistinct = [a1, a2, a3, a4]*

*// отсортировать значения по алфавиту в обратном порядке*

List<String> sortedReverse = collection.stream().sorted((o1, o2) -> -o1.compareTo(o2)).collect(Collectors.toList());

System.out.println("sortedReverse = " + sortedReverse);

*// напечатает sortedReverse = [a4, a4, a3, a2, a1, a1]*

*// отсортировать значения по алфавиту в обратном порядке и убрать дубликаты*

List<String> distinctReverse = collection.stream().sorted((o1, o2) -> -o1.compareTo(o2)).distinct().collect(Collectors.toList()); System.out.println("distinctReverse = " + distinctReverse);

*// напечатает sortedReverse = [a4, a3, a2, a1]*

*// \*\*\*\*\*\*\*\*\*\*\*\* Работа с объектами*

*// Зададим коллекцию людей*

Collection<People> peoples = Arrays.asList( new People("Вася", 16, Sex.MAN), new People("Петя", 23, Sex.MAN), new People("Елена", 42, Sex.WOMEN), new People("Иван Иванович", 69, Sex.MAN) );

*// Отсортировать по имени в обратном алфавитном порядке*

Collection<People> byName = peoples.stream().sorted((o1,o2) -> -o1.getName().compareTo(o2.getName())).collect(Collectors.toList()); System.out.println("byName = " + byName);

*// byName = [{name='Петя', age=23, sex=MAN}, {name='Иван Иванович', age=69, sex=MAN}, {name='Елена', age=42, sex=WOMEN}, {name='Вася', age=16, sex=MAN}]*

*// Отсортировать сначала по полу, а потом по возрасту*

Collection<People> bySexAndAge = peoples.stream().sorted((o1, o2) -> o1.getSex() != o2.getSex() ? o1.getSex(). compareTo(o2.getSex()) : o1.getAge().compareTo(o2.getAge())).collect(Collectors.toList()); System.out.println("bySexAndAge = " + bySexAndAge);

*// bySexAndAge = [{name='Вася', age=16, sex=MAN}, {name='Петя', age=23, sex=MAN}, {name='Иван Иванович', age=69, sex=MAN}, {name='Елена', age=42, sex=WOMEN}]*

}

private enum Sex {

MAN, WOMEN }

private static class People {

private final String name;

private final Integer age;

private final Sex sex;

public People(String name, Integer age, Sex sex)

{ this.name = name;

this.age = age;

this.sex = sex; }

public String getName() {

return name;

}

public Integer getAge() {

return age;

}

public Sex getSex() {

return sex;

}

@Override public String toString() {

return "{" + "name='" + name + '\'' + ", age=" + age + ", sex=" + sex + '}';

}

@Override public boolean equals(Object o) {

if (this == o) return true;

if (!(o instanceof People)) return false;

People people = (People) o;

return Objects.equals(name, people.name) && Objects.equals(age, people.age) && Objects.equals(sex, people.sex); }

@Override public int hashCode() { return Objects.hash(name, age, sex); } }

### Примеры использования Max и Min функций

**Условие**: дана коллекция строк Arrays.asList(«a1», «a2», «a3», «a1»), и коллекция класса Peoples из прошлых примеров

*// Метод max вернет максимальный элемент, в качестве условия использует компаратор*

*// Метод min вернет минимальный элемент, в качестве условия использует компаратор*

private static void testMinMax() {

*// \*\*\*\*\*\*\*\*\*\*\*\* Работа со строками*

Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");

*// найти максимальное значение*

String max = collection.stream().max(String::compareTo).get();

System.out.println("max " + max);

*// напечатает a3*

*// найти минимальное значение*

String min = collection.stream().min(String::compareTo).get();

System.out.println("min " + min);

*// напечатает a1*

*// \*\*\*\*\*\*\*\*\*\*\*\* Работа со сложными объектами*

*// Зададим коллекцию людей*

Collection<People> peoples = Arrays.asList( new People("Вася", 16, Sex.MAN), new People("Петя", 23, Sex.MAN), new People("Елена", 42, Sex.WOMEN), new People("Иван Иванович", 69, Sex.MAN) );

*// найти человека с максимальным возрастом*

People older = peoples.stream().max((p1, p2) -> p1.getAge().compareTo(p2.getAge())).get();

System.out.println("older " + older);

*// напечатает {name='Иван Иванович', age=69, sex=MAN}*

*// найти человека с минимальным возрастом*

People younger = peoples.stream().min((p1, p2) -> p1.getAge().compareTo(p2.getAge())).get(); System.out.println("younger " + younger); *// напечатает {name='Вася', age=16, sex=MAN}* }

**Примеры использования ForEach и Peek функций**

Обе ForEach и Peek по сути делают одно и тоже, меняют свойства объектов в стриме, единственная разница между ними в том что **ForEach терминальная** и она заканчивает работу со стримом, в то время как **Peek конвейерная** и работа со стримом продолжается.

*// Метод ForEach применяет указанный метод к каждому элементу стрима и заканчивает работу со стримом*

private static void testForEach() {

Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");

*// Напечатать отладочную информацию по каждому элементу стрима* System.out.print("forEach = "); collection.stream().map(String::toUpperCase).forEach((e) -> System.out.print(e + ","));

*// напечатает forEach = A1,A2,A3,A1,*

Collection<StringBuilder> list = Arrays.asList(new StringBuilder("a1"), new StringBuilder("a2"), new StringBuilder("a3"));

list.stream().forEachOrdered((p) -> p.append("\_new")); System.out.println("forEachOrdered = " + list);

*// напечатает forEachOrdered = [a1\_new, a2\_new, a3\_new]* }

*// Метод Peek возвращает тот же стрим, но при этом применяет указанный метод к каждому элементу стрима*

private static void testPeek() {

Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");

*// Напечатать отладочную информацию по каждому элементу стрима*

System.out.print("peak1 = ");

List<String> peek = collection.stream().map(String::toUpperCase).peek((e) -> System.out.print(e + ",")). collect(Collectors.toList());

System.out.println();

*// напечатает peak1 = A1,A2,A3,A1,*

System.out.println("peek2 = " + peek);

*// напечатает peek2 = [A1, A2, A3, A1]*

Collection<StringBuilder> list = Arrays.asList(new StringBuilder("a1"), new StringBuilder("a2"), new StringBuilder("a3"));

List<StringBuilder> newList = list.stream().peek((p) -> p.append("\_new")).collect(Collectors.toList()); System.out.println("newList = " + newList);

*// напечатает newList = [a1\_new, a2\_new, a3\_new]* }

### Примеры использования Reduce функции - хитрости

Метод reduce позволяет выполнять агрегатные функции на всей коллекцией (такие как сумма, нахождение минимального или максимального значение и т.п.), он возвращает одно значение для стрима, функция получает два аргумента — значение полученное на прошлых шагах и текущее значение.  
  
**Условие**: Дана коллекция чисел Arrays.asList(1, 2, 3, 4, 2) + лист с People

*// Метод reduce позволяет выполнять агрегатные функции на всей коллекцией (такие как сумма, нахождение минимального или максимального значение и т.п.)* *// Он возвращает одно Optional значение*

*// map - преобразует один объект в другой (например, класс одного тип в другой)*

*// mapToInt - преобразование объектов в числовой стрим (стрим, состоящий из значений int)*

private static void testReduce() {

*// \*\*\*\*\*\*\*\*\*\*\*\* Работа с числовыми объектами*

Collection<Integer> collection = Arrays.asList(1, 2, 3, 4, 2);

*// Вернуть сумму*

Integer sum = collection.stream().reduce((s1, s2) -> s1 + s2).orElse(0);

*// 12*

*Вернуть максимум*

Integer max1 = collection.stream().reduce((s1, s2) -> s1 > s2 ? s1 : s2).orElse(0);

*// или*

Integer max2 = collection.stream().reduce(Integer::max).orElse(0);

*// 4*

*Вернуть последний элемент*

Integer last = collection.stream().reduce((s1, s2) -> s2).orElse(0);

*// 2*

*// Вернуть сумму чисел, которые больше 2*

Integer sumMore2 = collection.stream().filter(o -> o > 2).reduce((s1, s2) -> s1 + s2).orElse(0);

*// 7*

*// Вернуть сумму нечетных чисел*

Integer sumOdd = collection.stream().filter(o -> o % 2 != 0).reduce((s1, s2) -> s1 + s2).orElse(0);

*// 4*

*// \*\*\*\*\*\*\*\*\*\*\*\* Работа со сложными объектами*

*// Зададим коллекцию людей*

Collection<People> peoples = Arrays.asList( new People("Вася", 16, Sex.MAN), new People("Петя", 23, Sex.MAN), new People("Елена", 42, Sex.WOMEN), new People("Иван Иванович", 69, Sex.MAN) );

*// Найдем самого старшего мужчину*

int oldMan = peoples.stream().filter((p) -> p.getSex() == Sex.MAN).map(People::getAge).reduce((s1, s2) -> s1 > s2 ? s1 : s2).get();

*// напечатает 69*

*// Найдем самого минимальный возраст человека у которого есть буква е в имени*

int younger = peoples.stream().filter((p) -> p.getName().contains("е")).mapToInt(People::getAge).reduce((s1, s2) -> s1 < s2 ? s1 : s2).orElse(0); System.out.println("younger = " + younger);

*// напечатает 23*

}

### Примеры использования toArray и collect функции

Если с toArray все просто, можно либо вызвать toArray() получить Object[], либо toArray(T[]::new) — получив массив типа T, то collect позволяет много возможностей преобразовать значение в коллекцию, map'у или любой другой тип. Для этого используются статические методы из Collectors, например преобразование в List будет stream.collect(Collectors.toList()).

*// Метод collect преобразует stream в коллекцию или другую структуру данных*

*// Полезные статические методы из Collectors:*

*// toList, toCollection, toSet - представляют стрим в виде списка, коллекции или множества*

*// toConcurrentMap, toMap - позволяют преобразовать стрим в map, используя указанные функции*

*// averagingInt, averagingDouble, averagingLong - возвращают среднее значение* *// summingInt, summingDouble, summingLong - возвращает сумму*

*// summarizingInt, summarizingDouble, summarizingLong - возвращают SummaryStatistics с разными агрегатными значениями*

*// partitioningBy - разделяет коллекцию на две части по соответствию условию и возвращает их как Map<Boolean, List>*

*// groupingBy - разделить коллекцию по условию и вернуть Map<N, List<T>>, где T - тип последнего стрима, N - значение разделителя*

*// mapping - дополнительные преобразования значений для сложных Collector'ов*

private static void testCollect()

*// \*\*\*\*\*\*\*\* Работа со строками*

Collection<String> strings = Arrays.asList("a1", "b2", "c3", "a1");

*// Получение списка из коллекции строк без дубликатов*

List<String> distinct = strings.stream().distinct().collect(Collectors.toList()); System.out.println("distinct = " + distinct);

*// напечатает distinct = [a1, b2, c3]*

*// Получение массива уникальных значений из коллекции строк*

String[] array = strings.stream().distinct().map(String::toUpperCase).toArray(String[]::new); System.out.println("array = " + Arrays.asList(array));

*// напечатает array = [A1, B2, C3]*

*// Объединить все элементы в одну строку через разделитель : и обернуть тегами <b> ... </b>*

String join = strings.stream().collect(Collectors.joining(" : ", "<b> ", " </b>")); System.out.println("join = " + join);

*// напечатает <b> a1 : b2 : c3 : a1 </b>*

*// Преобразовать в map, где первый символ ключ, второй символ значение*

Map<String, String> map = strings.stream().distinct().collect(Collectors.toMap((p) -> p.substring(0, 1), (p) -> p.substring(1, 2)));

System.out.println("map = " + map);

*// напечатает map = {a=1, b=2, c=3}*

*// Преобразовать в map, сгруппировав по первому символу строки*

Map<String, List<String>> groups = strings.stream().collect(Collectors.groupingBy((p) -> p.substring(0, 1)));

System.out.println("groups = " + groups);

*// напечатает groups = {a=[a1, a1], b=[b2], c=[c3]}*

*// Преобразовать в map, сгруппировав по первому символу строки и в качестве значения взять второй символ объединим через :*

Map<String, String> groupJoin = strings.stream().collect(Collectors.groupingBy((p) -> p.substring(0, 1), Collectors.mapping((p) -> p.substring(1, 2), Collectors.joining(":"))));

System.out.println("groupJoin = " + groupJoin);

*// напечатает groupJoin = groupJoin = {a=1/1, b=2, c=3}*

*// \*\*\*\*\*\*\*\* Работа с числами*

Collection<Integer> numbers = Arrays.asList(1, 2, 3, 4);

*// Получить сумму нечетных чисел*

long sumOdd = numbers.stream().collect(Collectors.summingInt(((p) -> p % 2 == 1 ? p : 0)));

System.out.println("sumOdd = " + sumOdd);

*// напечатает sumEven = 4*

*// Вычесть у каждого элемента 1 и получить среднее*

double average = numbers.stream().collect(Collectors.averagingInt((p) -> p - 1)); System.out.println("average = " + average);

*// напечатает average = 1.5*

*// Прибавить к числам 3 и получить статистику*

IntSummaryStatistics statistics = numbers.stream().collect(Collectors.summarizingInt((p) -> p + 3)); System.out.println("statistics = " + statistics);

*// напечатает statistics = IntSummaryStatistics{count=4, sum=22, min=4, average=5.500000, max=7}*

*// Получить сумму четных чисел через IntSummaryStatistics*

long sumEven = numbers.stream().collect(Collectors.summarizingInt((p) -> p % 2 == 0 ? p : 0)).getSum();

System.out.println("sumEven = " + sumEven);

*// напечатает sumEven = 6*

*// Разделить числа на четные и нечетные* Map<Boolean, List<Integer>> parts = numbers.stream().collect(Collectors.partitioningBy((p) -> p % 2 == 0)); System.out.println("parts = " + parts);

*// напечатает parts = {false=[1, 3], true=[2, 4]}* }

+ Есть возможность создания собственного Collector

Кроме Collector'ов уже определенных в Collectors можно так же создать собственный Collector

- если нужно будет когда-то, то посмотреть

\_\_\_\_\_\_\_\_\_

Задачи по стримам:

1. Перевернуть в map values и keys:

public class MapReverEx {  
  
 public static void main(String[] args) {  
   
 Map<Integer, String> mymap = Map.*of*(1, "one", 2, "two", 3, "three");  
   
 Map<String, Integer> newmap =  
 mymap.entrySet().stream().collect(Collectors.*toMap*(x->x.getValue(), y->y.getKey()));  
   
 System.*out*.println(newmap);  
 }  
}

1. Фибоначчи первые 10 чисел

public class FibonacciStreamEx {  
 public static void main(String[] args) {  
  
 Stream.*iterate*(new int [] {0,1}, arr-> new int [] {arr[1], arr[0]+arr[1]})  
 .limit(10).map(x->x[0])  
 .forEach(System.*out*::println);  
 }  
}