Lab: Stream API

Problems for exercises and homework for the "Java Advanced" course @ SoftUni.

You can check your solutions here: https://judge.softuni.bg/Contests/1040/Stream-API-Lab.

Part I: Stream<T> and Types of Streams

1. Take Two

Read a **sequence of integers**, given on a single line separated by a space.

Finds all **unique** elements, such that **10 ≤ n ≤ 20** and print only **the first 2 elements**.

If there are **fewer than 2 elements**, print as much as there are. If there are **no elements**, print nothing.

Examples

Input	Output
15 2 15 14 12	15 14
17 -2 3	17
-2 3	*(no output)*

Hints

- Read the input using a **`Scanner`** or a **`BufferedReader`** and parse the strings to a list of numbers:

```
Scanner scanner = new Scanner(System.in);
List<String> tokens =
        Arrays.asList(scanner.nextLine().split("\\s+"));

List<Integer> numbers = new ArrayList<>();
for (String token : tokens) {
    numbers.add(Integer.valueOf(token));
}
```

- Filter the numbers with **`filter()`**, take the unique ones with **`distinct()`**, take only two from the stream with **`limit()`** and iterate over them while printing with **`forEach()`**:

```
numbers.stream()
        .filter(n -> 10 <= n && n <= 20)
        .distinct()
        .limit(2)
        .forEach(n -> System.out.print(n + " "));
```

2. Upper Strings

Read a sequence of strings, given on a single line separated with a space.

Map each to upper case and print them, using the Stream API.

Examples

Input	Output
Pesho Gosho Stefan	PESHO GOSHO STEFAN
Soft Uni Rocks	SOFT UNI ROCKS
(empty line)	*(no output)*

Hints

- Read the input using a **`Scanner`** or a **`BufferedReader`** into a list of strings **`List<String>`**:

```
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
List<String> strings = Arrays.asList(reader.readLine().split("\\s+"));
```

- Call a stream over the list and map every element to upper case. Iterate over the stream and print the result:

```
strings.stream()
        .map(s -> s.toUpperCase())
        .forEach(s -> System.out.print(s + " "));
```

3. First Name

Read a **sequence of names**, given on a single line, separated by a space.

Read a **sequence of letters**, given on the next line, separated by a space.

Find the **names that start with one of the given letters** and print the first of them (**ordered lexicographically**).

If there is **no name** that conforms to the requirement, **print "No match"**.

Examples

Input	Output
Rado Plamen Gosho p r	Plamen
Plamen Gosho Rado s c	No match

Hints

- You can use a hash set to store letter, for efficient searching
- Make sure you are comparing letter with the same casing (lower or upper)
- Use **`filter()`**, **`sorted()`** and **`findFirst()`**
- Use **`Optional<T>`**

4. Average of Doubles

Read a **sequence of double numbers**, given on a single line, separated by a space.

Find the **average of all elements**, using the Stream API.

Round the output to the second digit after the decimal separator.

If there are **no numbers** in the sequence, **print "No match"**.

Examples

Input	Output
3 4 5 6	4.50
3.14 5.2 6.18	4.84
(empty list)	No match

Hints

- Use a primitive stream **`DoubleStream`**
- Use **`OptionalDouble`**
- Make sure to filter empty strings before transforming the stream

Part II: Types of Operations

5. Min Even Number

Read a sequence of numbers, given on a single line, separated by a space.

Find the smallest number of all even numbers, using the Stream API.

If there are **no numbers** in the sequence, **print "No match"**.

Examples

Input	Output
1 2 3 4 5 6	2.00
3.14 -2.00 1.33	-2.00
(empty list)	No match

Hints

- Use map function to map the objects to **`Double`**
- Make sure to filter empty strings
- Filter the even numbers
- Get the smallest number using **`Double.compare(x1, x2)`**

6. Find and Sum Integers

Read a sequence of elements, given on a single line, separated by a space.

Filter all elements that are integers and calculate their sum, using the Stream API.

If there are **no numbers** in the sequence, **print "No match"**.

Examples

Input	Output
Sum 3 and 4	7

Sum -3 and -4	-7
Sum three and four	No match

Hints

- Use **filter → map → reduce** pattern
- Check if element's char at index 0 is a sign (+ or -)
- Check if all else element's chars are digits

7. *Map Districts

On the first line, you are given the population count of districts in different cities, separated by a single space in the format **"city:district population"**.

On the second line, you are given the minimum population for filtering of the towns. The **population of a town** is the **sum of populations of all of its districts**.

Print all **cities** with population greater than a given. **Sort cities and districts** by descending population and **print top 5 districts for a given city**.

For a better understanding, see the examples below.

Examples

Input	Output
Pld:9 Pld:13 Has:7 Sof:20 Sof:10 Sof:15 10	Sof: 20 15 10 Pld: 13 9
Sof:10 Sof:12 Sof:15 10	Sof: 15 12 10
Sof:5 15	*(no output)*

Hints

- Read the input into a **proper collection**:

```
HashMap<String, List<Integer>> cities = new HashMap<>();
List<String> tokens = Arrays.asList(scanner.nextLine().split("\\s+"));

for (String token : tokens) {
    String[] tokenArgs = token.split(":");
    String city = tokenArgs[0];
    int districtPopulation = Integer.valueOf(tokenArgs[1]);

    cities.putIfAbsent(city, new ArrayList<>());
    cities.get(city).add(districtPopulation);
}
```

- Read the population bound

```
int bound = Integer.valueOf(scanner.nextLine());
```

- **Filter**, **sort** and **print** the cities:

```
cities.entrySet().stream()
        .filter(getFilterByPopulationPredicate(bound))
        .sorted(getSortByDescendingPopulationComparator())
        .forEach(getPrintMapEntryConsumer());
```

- Crete methods for generating **lambda expressions**, stored in **functional interfaces**
- Crete a method that returns a **predicate for filtering**:

```
private static Predicate<Map.Entry<String, List<Integer>>> getFilterByPopulationPredicate(int bound) {
    return kv -> kv.getValue().stream()
            .mapToInt(Integer::valueOf)
            .sum() >= bound;
}
```

- Create a method that returns a **comparator for sorting:**

```
private static Comparator<Map.Entry<String, List<Integer>>> getSortByDescendingPopulationComparator() {
    return (kv1, kv2) ->
            Integer.compare(
                    kv2.getValue().stream().mapToInt(Integer::valueOf).sum(),
                    kv1.getValue().stream().mapToInt(Integer::valueOf).sum());
}
```

- Create a method that returns a **consumer for printing a map entry:**

```
private static Consumer<Map.Entry<String, List<Integer>>> getPrintMapEntryConsumer() {
    return kv -> {
        System.out.print(kv.getKey() + ": ");
        kv.getValue().stream()
                .sorted((s1, s2) -> s2.compareTo(s1))
                .limit(5)
                .forEach(dp -> System.out.print(dp + " "));
        System.out.println();
    };
}
```

8. Bounded Numbers

On the first line, read two numbers, a **lower** and **an upper bound**, separated by a space.

On the second line, read a sequence of numbers, separated by a space.

Print all numbers, such that **`[lower bound] ≤ n ≤ [upper bound]`**.

Examples

Input	Output
5 7 1 2 3 4 5 6 7 8 9	5 6 7
7 5 9 5 7 2 6 8	5 7 6
3 4 5 6 7 8	*(no output)*

Hints

- Use **`collect(Collectors.toList())`**