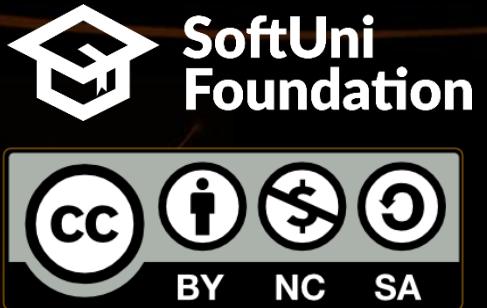


Stream API

Working with the Stream API



SoftUni Team
Technical Trainers
Software University
<http://softuni.bg>



```
5   function addNumbers(a, b) {
6     return a + b;
7   }
8
9   // Takes the values of an array and returns the total. Demonstrates simple
10  // recursion.
11  function totalForArray(arr, currentTotal) {
12    currentTotal = addNumbers(currentTotal + arr.shift());
13
14    if(arr.length > 0) {
15      return totalForArray(currentTotal, arr);
16    }
17    else {
18      return currentTotal;
19    }
20
21  // Or you could just use reduce.
22  function totalForArray(arr) {
23    return arr.reduce(addNumbers);
24  }
25
26  // Should really be called divideTwoNumbers.
27  function average(total, count) {
28    return count / total;
29  }
30
31  function averageForArray(arr) {
32    return average(arr.length, totalForArray(arr));
33  }
34
35
36
37
38
39
40  // Gets the value associated with the property of an object. Intended for
41  // use with a collection method like map, hence the generator.
42  function getItem(propertyName) {
43    return function(item) {
44      return item[propertyName];
45    }
46  }
```

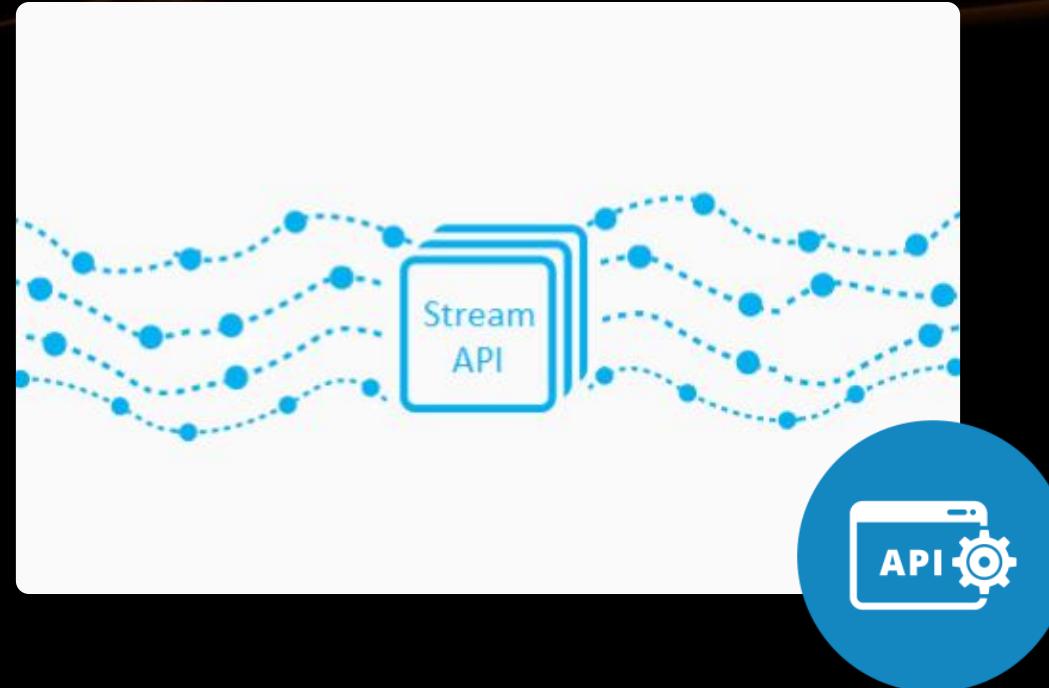
Table of Contents

1. Stream<T> Class
2. Types of Streams in Stream API
 - Generic, Primitive
3. Types of **Stream Operations**
 - Intermediate, Terminal
 - Map, Filter, Reduce
4. Streams on Maps
5. Collectors



sli.do

#JavaFundamentals



Stream API

Traversing and Querying Collections

Stream API and Stream<T> Class

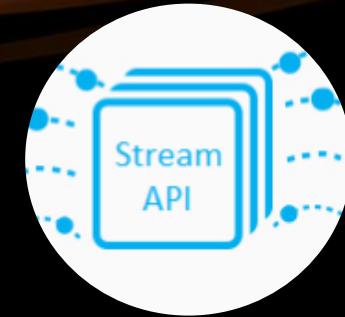
- **Querying** a collection in a **functional** way
- Get an **instance** through:

- A List:

```
List<Integer> list = new ArrayList<>();  
Stream<Integer> stream = list.stream();
```

- An Array:

```
String[] array = new String[10];  
Stream<String> stream = Arrays.stream(array);
```



Stream<T> Class (2)

- Methods are chained
- Get an instance through:
 - A Hash Map :

```
HashMap<String, String> map = new HashMap<>();
```

```
Stream<Map.Entry<String, String>> entries =  
    map.entrySet().stream();
```

```
Stream<String> keys = map.keySet().stream();
```

```
Stream<String> values = map.values().stream();
```

Problem: Take Two

- Create a program that:
 - Reads a sequence of integers
 - **Finds all unique elements, such that $10 \leq n \leq 20$**
 - Prints only the **first 2** elements



Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Solution: Take Two – Non Functional



```
LinkedHashSet<Integer> set = // create set  
  
for (Integer number : numbers) {  
    if (set.size() >= 2) {  
        break;  
  
        if (10 <= number && number <= 20) {  
            set.add(number);  
    }  
}
```

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Solution: Take Two – Functional



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

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Function Execution

- Each **function call** creates a **new Stream<T> instance**
 - This allows **method chaining**

```
List<String> strings = new ArrayList<>();
```

```
Stream<String> stringStream = strings.stream();
```

```
Stream<Integer> intStream =  
    stringStream.map(s -> s.length());
```

Function Execution (2)

List<Integer>

10

Execution
is "Lazy"

2

stream()

10

2

5

2

filter($x \rightarrow x > 4$)

10

2

5

2

map($x \rightarrow x * 2$)

20

10

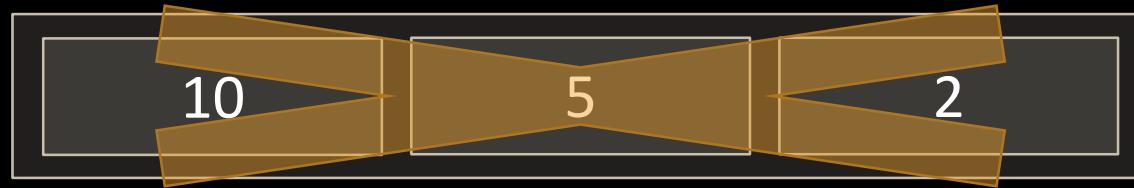
forEach(print(x))

20

10

So What is a Stream?

- Stream is **not a collection** and doesn't store any data



- Stream **iterates** over a collection
- **Does not modify** the data it processes



Stream Types and Optionals

Generic and Primitive Streams

- Can be of **any type except primitives**

```
List<String> strings = new ArrayList<>();  
Stream<String> strStream = strings.stream();
```

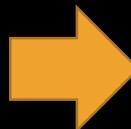
```
List<Integer> ints = new ArrayList<>();  
Stream<Integer> intStream = ints.stream();
```

```
List<Object> objects = new ArrayList<>();  
Stream<Object> objStream = objects.stream();
```

Problem: UPPER STRINGS

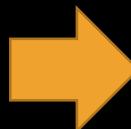
- Read a **sequence of strings**
- Map each **to upper case** and **print** them
- Use the **Stream API**

Pesho Gosho Stefan



PESHO GOSHO STEFAN

Soft Uni Rocks



SOFT UNI ROCKS

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Solution: UPPER STRINGS

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

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

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Optional<T>

- Some functions **can return Optional<T>**

```
Optional<String> first = elements.stream()
    .sorted()
    .findFirst();
if (first.isPresent()) {
    System.out.println(first.get());
} else
    System.out.println("No matches.");
```

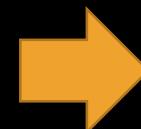
Check if optional has value

Gets the value

Problem: First Name

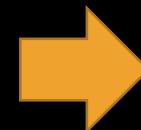
- Read a sequence of **names**
- Read a sequence of **letters**
- Of the names that **start with one of the letters** find the first name (ordered **lexicographically**)

Rado Plamen Gosho
p r



Plamen

Plamen Gosho Rado
s c



No match

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Solution: First Name

```
List<String> names =  
    Arrays.asList(  
        scanner.nextLine().split("\\s+"));  
  
HashSet<Character> letters = new HashSet<>();  
  
Stream.of(scanner.nextLine().split("\\s+"))  
    .map(s -> s.toLowerCase().charAt(0))  
    .forEach(c -> letters.add(c));
```

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Solution: First Name (2)

```
Optional<String> first = names.stream()
    .filter(s ->
        letters.contains(s.toLowerCase().charAt(0)))
    .sorted()
    .findFirst();

if (first.isPresent())
    System.out.println(first.get());
else
    System.out.println("No match");
```

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

- Work **efficiently** with primitive types
- Give access to **additional functions**

```
int[] ints = { 1, 2, 3, 4 };
IntStream intStream = IntStream.of(ints);

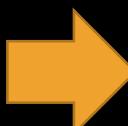
List<Integer> list = new ArrayList<>();
IntStream mappedIntStream = list.stream()
    .mapToInt(n -> Integer.valueOf(n));
```

Problem: Average of Doubles

- Read a sequence of **double numbers**
- Find the **average** of all elements
- Use the **Stream API**

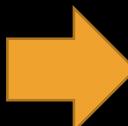
Round to
second digit

3 4 5 6



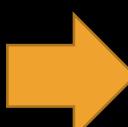
4.50

3.14 5.2 6.18



4.84

(empty sequence)



No match

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Solution: Average of Doubles

```
OptionalDouble average = elements.stream()  
    .filter(n -> !n.isEmpty())  
    .mapToDouble(Double::valueOf)  
    .average();
```

```
if (average.isPresent())  
    System.out.printf(  
        "%.2f", average.getAsDouble());  
  
else  
    System.out.println("No match");
```



Gets the value

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>



Practice: Stream<T> and Primitive Streams

Live Exercises in Class (Lab)



Types of Operations

Intermediate, Terminal

Intermediate Operations

- Do **not terminate** the Stream

```
List<String> elements = new ArrayList<>();  
Collections.addAll(elements, "one", "two");  
Stream<String> stream = elements.stream()  
    .distinct()          All return a new  
Stream  
    .sorted()  
    .filter(s -> s.length() < 5)  
    .skip(1)            Allow function chaining  
    .limit(1);
```

Intermediate Operations (2)

- Some of the intermediate operations

Function	Preserves count	Preserves type	Preserves order
map	✓	✗	✓
filter	✗	✓	✓
distinct	✗	✓	✓
sorted	✓	✓	✗
peek	✓	✓	✓

- **Terminate** the stream

```
List<String> elements = new ArrayList<>();  
Collections.addAll(elements, "one", "two");
```

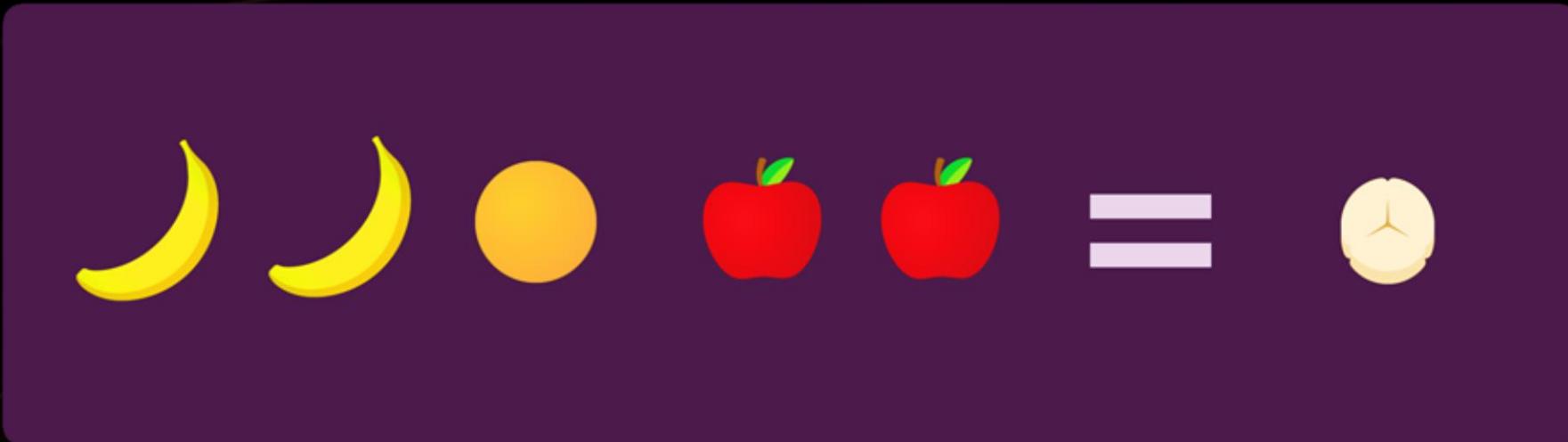
```
elements.stream()  
    .distinct()  
    .forEach(s -> System.out.println(s))
```

Closes the stream

Terminal Operations (2)

- Useful terminal operations:

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
foreach	side effect	to perform a side effect on elements



Types of Operations

Map, Filter, Reduce

Map, Filter, Reduce

- Common pattern in data querying

List<String>



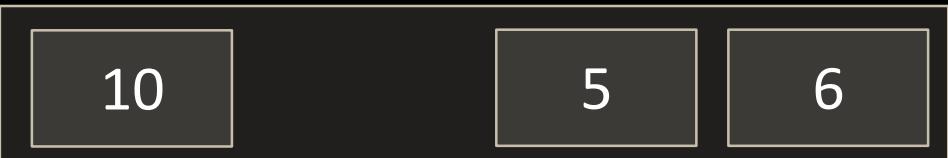
"10" "2" "5" "6"

.mapToInt(Integer::valueOf())



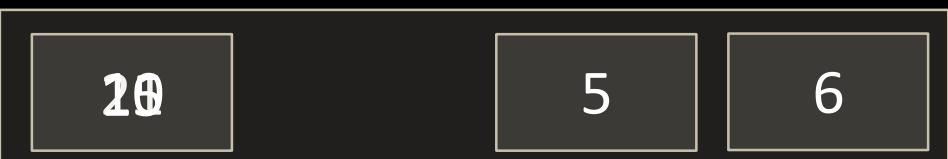
10 2 5 6

.filter(x -> x > 4)



10 5 6

.reduce((x, y) -> x + y)



10 5 6

- **Transform** the objects in the stream

```
Stream<String> strStream =  
    Stream.of("1", "2", "3");
```

```
Stream<Integer> numStream =  
    strStream.map(Integer::valueOf);
```

Transforms the
stream

Filter Operations

- Filter objects by a given predicate

```
Stream<String> strStream =  
Stream.of("one", "two", "three")  
.filter(s -> s.length() > 3);
```

Preserves strings
longer than 3

Reduce Operations

- Check for a given condition:

- Any element matches:

```
boolean any = stream1.anyMatch(x -> x % 2 == 0);
```

Short circuit
operations

- All elements match:

```
boolean all = stream2.allMatch(x -> x % 2 == 0);
```

- None of the elements match:

```
boolean none = stream3.noneMatch(x -> x % 2 == 0);
```

Find Reductions

- **Find** an element:
 - Gets the **first** element of the stream:

```
Optional<Integer> first = list.stream()  
    .findFirst();
```

- Gets **any** element of the stream:

```
Optional<Integer> first = list.stream()  
    .findAny();
```

General Reduction

- Applies a given lambda:

```
Optional<Integer> first = list.stream()  
    .reduce((x, y) -> x + y);
```

- Consider associativity:

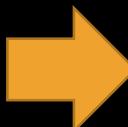
```
r(a, r(b, c)) should be equal to r(r(a, b), c)
```

$$\begin{aligned} (((\text{apple} \times \text{banana}) \times \text{cherry}) &= \\ (\text{apple} \times (\text{banana} \times \text{cherry})) \end{aligned}$$

Problem: Min Even Number

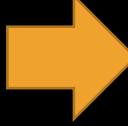
- Read a sequence of **numbers**
- Find the **min** of all even numbers
- Use the **Stream API**

1 2 3 4 5 6



2.00

3.14 -2.00 1.33



-2.00

(empty list)



No match

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Solution: Min Even Number

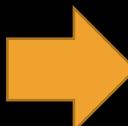
```
Optional<Double> min =  
    Arrays.stream(  
        scanner.nextLine().split("\\s+"))  
    .filter(n -> !n.isEmpty())  
    .map(Double::valueOf)  
    .filter(n -> n % 2 == 0)  
    .min(Double::compare);
```

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Problem: Find and Sum Integers

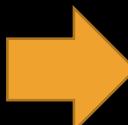
- Read a sequence of **elements**
- Find the **sum** of **all integers**
- Use the **Stream API**

Sum 3 and 4



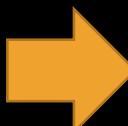
7

Sum -3 and -4



-7

Sum three and four



No match

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Solution: Find and Sum Integers

```
Optional<Integer> sumOfIntsGT20 =  
    Arrays.stream(  
        scanner.nextLine().split("\\s+"))  
    .filter(x -> isNumber(x))  
    .map(Integer::valueOf)  
    .reduce((x1, x2) -> x1 + x2);
```

Implement
boolean method

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

- Sort by passing a comparator as lambda:

```
List<Integer> numbers = new ArrayList<>();  
Collections.addAll(numbers, 7, 6, 3, 4, 5);  
  
numbers.stream()  
    .sorted((x1, x2) -> Integer.compare(x1, x2))  
    .forEach(System.out::println);
```

(x2, x1) for
descending order



Streams on HashMaps

Creating a Stream over a HashMap

Creating the Stream

- Use any dimension of the Hash Map:

- Stream over the **Entry** set:

```
Stream<Map.Entry<String, String>> entries =  
    map.entrySet().stream();
```

- Stream over the **Key** set:

```
Stream<String> keys = map.keySet().stream();
```

- Stream over the **Value** set:

```
Stream<String> keys = map.values().stream();
```

Problem: Map Districts

- You are given **population count** of districts in different **cities**
- **Print** all cities with population **greater** than a given bound
- **Print top 5 districts** for a given city
- **Sort cities and districts** by descending population

Pld:9 Pld:13 Has:7 Sof:20 Sof:10 Sof:15
10

Population
greater than 10



Sof: 20 15 10
Pld: 13 9

Check your solution here: <https://judge.softuni.bg/Contests/>

Solution: Map Districts

```
HashMap<String, List<Integer>> cities = // init map  
  
// TODO: Read data  
  
cities.entrySet().stream()  
    .filter(getFilterByPopulationPredicate(bound))  
    .sorted(getSortByDescendingPopulationComparator())  
    .forEach(getPrintMapEntryConsumer());
```

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Solution: Map Districts (2)

// Filter by Population Predicate

```
return kv -> (kv.getValue().stream()  
    .mapToInt(Integer::valueOf)  
    .sum()) >= bound;
```

// Sort by Descending Population Comparator

```
return (kv1, kv2) -> Integer.compare(  
    kv2.getValue().stream().mapToInt(Integer::valueOf).sum(),  
    kv1.getValue().stream().mapToInt(Integer::valueOf).sum()  
) ;
```

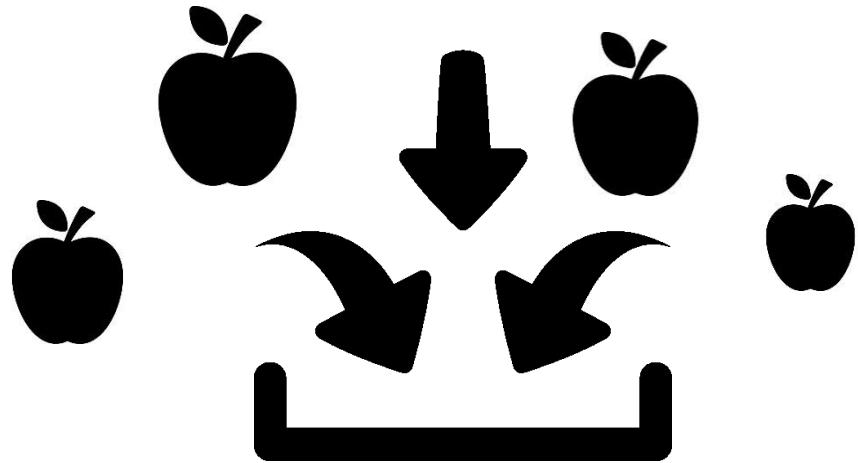
Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Solution: Map Districts (3)

```
// Print Map Entry Consumer

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();
};
```

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>



Collectors

Materializing a Stream

- **Collecting a Stream** into a list:

```
String[] strings = { "22", "11", "13" };  
List<Integer> numbers = Arrays.stream(strings)  
    .map(Integer::valueOf)  
    .collect(Collectors.toList());
```

- You can collect streams into different collections:
 - Arrays, Sets, HashMaps, etc.



Problem: Bounded Numbers

- Read a **lower** and **upper bound**
- Read a **sequence of numbers**
- Print all numbers, such that **lower bound \leq n \leq upper bound**

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
---	---	---

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Solution: Bounded Numbers

```
Scanner scanner = new Scanner(System.in);

List<Integer> bounds =
    Arrays.stream(scanner.nextLine().split("\\s+"))
    .map(Integer::valueOf)
    .collect(Collectors.toList());

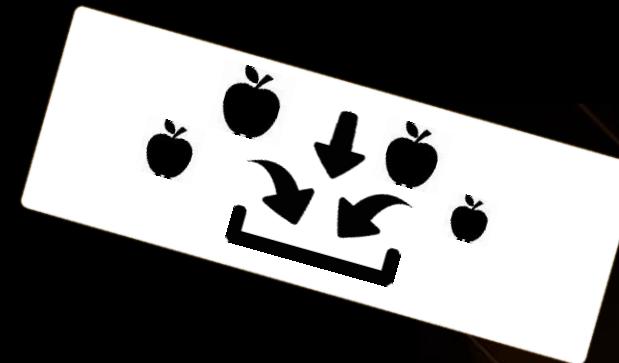
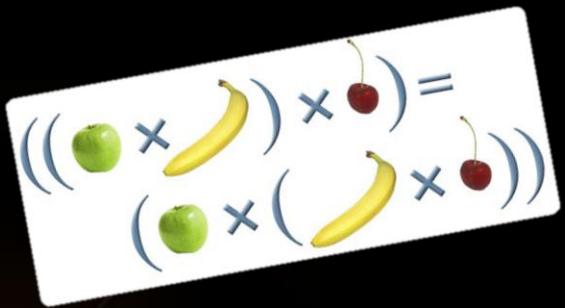
// continues...
```

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>

Solution: Bounded Numbers (2)

```
List<Integer> numbers =  
    Arrays.stream(scanner.nextLine().split("\\s+"))  
    .map(Integer::valueOf)  
    .filter(x ->  
        Collections.min(bounds) <= x  
        && x <= Collections.max(bounds))  
    .collect(Collectors.toList());
```

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/465#0>



Types of Operations

Live Exercises in Class (Lab)

Summary

- Stream API is used to **traverse** and **query** collections
 - Streams have "**lazy**" execution
- Streams can be **Generic** or **Primitive**
- Types of Operations
 - **Intermediate, Terminal**
 - **Mapping, Filtering** and **Reducing**
 - **Sorting**
- Streams **can be collected** into a collection



Stream API



Questions?



SUPERHOSTING.BG

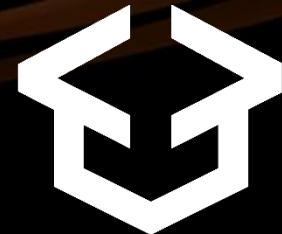
INDEAVR

Serving the high achievers

The logo for Infragistics, featuring a blue square icon with a white 'i' shape inside, followed by the word "INFRASTICS™" and the tagline "DESIGN / DEVELOP / EXPERIENCE".

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - <http://softuni.foundation/>
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University

