Part 10

Handling collections as streams

Learning Objectives

- You can handle collections using streams
- You know what a lambda-statement means
- You know the most common stream methods and are able to categorize them into intermediate and terminal operations.

Let's get to know collections, such as lists, as streams of values. Stream is a way of going through a collection of data such that the programmer determines the operation to be performed on each value. No record is kept of the index or the variable being processed at any given time.

With streams, the programmer defines a sequence of events that is executed for each value in a collection. An event chain may include dumping some of the values, converting values from one form to another, or calculations. A stream does not change the values in the original data collection, but merely processes them. If you want to retain the transformations, they need to be compiled into another data collection.

Let's begin to understand the usage of streams through a concrete example. Consider the following problem:

Write a program that reads input from a user and prints statistics about those inputs. When the user enters the string "end", the reading is stopped. Other inputs are numbers in string format. When you stop reading inputs, the program prints the number of positive integers divisible by three, and the average of all values.

```
// We initialise the scanner and the list into which the input is read
Scanner scanner = new Scanner(System.in);
List<String> inputs = new ArrayList<>();
```

```
// reading inputs
while (true) {
    String row = scanner.nextLine();
    if (row.equals("end")) {
        break;
    }

    inputs.add(row);
}

// counting the number of values divisible by three
long numbersDivisibleByThree = inputs.stream()
    .mapToInt(s -> Integer.valueOf(s))
    .filter(number -> number % 3 == 0)
    .count();

// working out the average
double average = inputs.stream()
    .mapToInt(s -> Integer.valueOf(s))
    .average()
    .getAsDouble();

// printing out the statistics
System.out.println("Divisible by three " + numbersDivisibleByThree);
System.out.println("Average number: " + average);
```

Let's take a closer look at the part of the program above where the inputs are dealt as streams.

```
// counting the number of values divisible by three
long numbersDivisibleByThree = inputs.stream()
    .mapToInt(s -> Integer.valueOf(s))
    .filter(number -> number % 3 == 0)
    .count();
```

A stream can be formed from any object that implements the Collection interface (e.g., ArrayList, HashSet, HashMap, ...) with the `stream()` method. The string values are then converted ("mapped") to integer form using the stream's `mapToInt(value -> conversion)` method. The conversion is implemented by the `valueOf` method of the `Integer` class, which we've used in the past. We then use the `filter (value -> filter condition)` method to filter out only those numbers that are divisible by three for further processing. Finally, we call the stream's `count()` method, which counts the number of elements in the stream and returns it as a `long` type variable.

Let's now look at the part of the program that calculates the average of the list elements.

```
// working out the average
double average = inputs.stream()
    .mapToInt(s -> Integer.valueOf(s))
    .average()
    .getAsDouble();
```

Calculating the average is possible from a stream that has the `mapToInt` method called on it. A stream of integers has an `average` method that returns an OptionalDouble-type object. The object has `getAsDouble()` method that returns the average of the list values as a `double` type variable.

A brief summary of the stream methods we've encountered so far.

Purpose and method	Assumptions
Stream formation: `stream()`	The method is called on collection that implements the Collection interface, such as an ArrayList Object. Something is done on the created stream.
Converting a stream into an integer stream: `mapToInt(value -> another)`	The stream transforms into one containing integers. A stream containing strings can be converted using, for instance, the valueOf method of the Integer class. Something is done with the stream containing integers.
Filtering values: `filter(value -> filter condition)`	The elements that do not satisfy the filter condition are removed from the string. On the right side of the arrow is a statement that returns a boolean. If the boolean is `true`, the element is accepted into the stream. If the boolean evaluates to false, the value is not accepted into the stream. Something is done with the filtered values.
Calculating the average: `average()`	Returns a OptionalDouble-type object that has a method `getAsDouble()` that returns a value of type `double`. Calling the method `average()` works on streams that contain integers - they can be created with the `mapToInt` method.
Counting the number of elements in a stream: `count()`	Returns the number of elements in a stream as a `long`-type value.

Programming exercise:

Average of Numbers

Points
1/1

Implement a program, which reads user input. If the user input is "end", the program stops reading input. The rest of the input is numbers. When the user input is "end", the program prints the average of all of the numbers.

Implement calculating the average using a stream!

Sample output

```
Input numbers, type "end" to stop.
2
4
6
end
average of the numbers: 4.0
```

Sample output

```
Input numbers, type "end" to stop.
-1
1
2
end
average of the numbers: 0.6666666666666666
```

Exercise submission instructions

How to see the solution

Programming exercise:

Average of selected numbers

Points 1/1

Implement a program, which reads user input. If the user input is "end", program stops reading input. The rest of the input is numbers.

Then user is asked if the program should print the average of all the positive numbers, or the average of all the negative numbers (n or p). If the user selects "n", the average of all the negative numbers is printed. Otherwise the average of all the positive numbers is printed.

Use streams to calculate the average and filter the numbers!

Sample output

```
Input numbers, type "end" to stop.
-1
1
2
end

Print the average of the negative numbers or the positive numbers? (n/p)
n
Average of the negative numbers: -1.0
```

Sample output

```
Input numbers, type "end" to stop.
-1
1
2
end

Print the average of the negative numbers or the positive numbers? (n/p)
p
Average of the positive numbers: 1.5
```

Exercise submission instructions

How to see the solution

Lambda Expressions

Stream values are handled by methods related to streams. Methods that handle values get a function as a parameter that determines what is done with each element. What the function does is specific to the method in question. For instance, the `filter` method used for filtering elements is provided a function which returns a boolean `true` or `false`, depending on whether to keep the value in the stream or not. The `mapToInt` method used for transformation is, on the other hand, provided a function which converts the value to an integer, and so on.

Why are the functions written in the form `value -> value > 5`?

The expression above, i.e., a *lambda expression*, is shorthand provided by Java for anonymous methods that do not have an "owner", i.e., they are not part of a class or an interface. The function contains both the parameter definition and the function body. The same function can be written in several different ways. See below.

```
// original
*stream*.filter(value -> value > 5).*furtherAction*

// is the same as
*stream*.filter((Integer value) -> {
    if (value > 5) {
        return true;
    }

    return false;
}).*furtherAction*
```

The same can be written explicitly so that a static method is defined in the program, which gets used within the function passed to the stream as a parameter.

```
public class Screeners {
    public static boolean greaterThanFive(int value) {
        return value > 5;
    }
}
```

```
// original
*stream*.filter(value -> value > 5).*furtherAction*

// is the same as
*stream*.filter(value -> Screeners.greaterThanFive(value)).*furtherAction*
```

The function can also be passed directly as a parameter. The syntax found below `Screeners::greaterThanFive` is saying: "use the static `greaterThanFive` method that's in the `Screeners` class".

```
// is the same as
*stream*.filter(Screeners::greaterThanFive).*furtherAction*
```

Functions that handle stream elements cannot change values of variables outside of the function. This has to do with how static methods behave - during a method call, there is no access to any variables outside of the method. With functions, the values of variables outside the function can be read, assuming that those values of those variables do not change in the program.

The program below demonstrates the situation in which a function attempts to make use of a variable outside the function. It doesn't work.

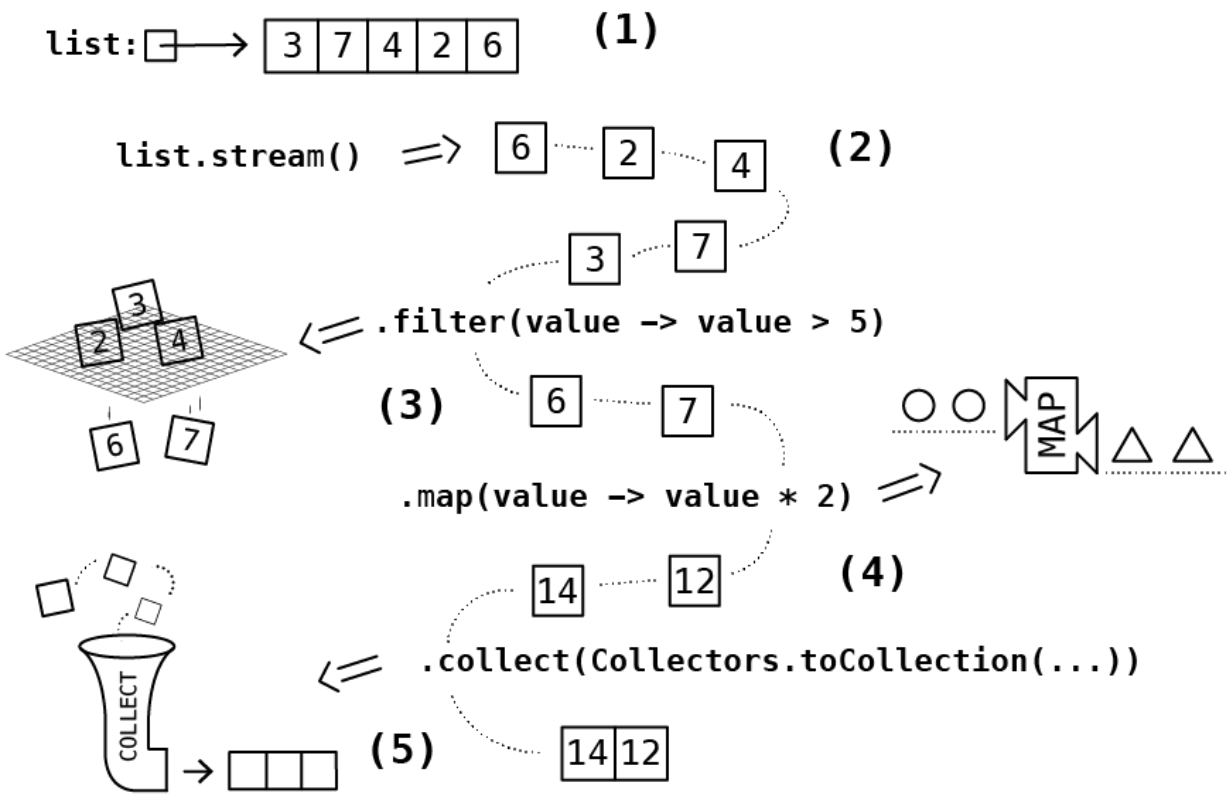
```
// initializing a scanner and a list to which values are read
Scanner scanner = new Scanner(System.in);
List<String> inputs = new ArrayList<>()

// reading inputs
while (true) {
    String row = scanner.nextLine();
    if (row.equals("end")) {
        break;
    }

    inputs.add(row);
```

```
}

int numberOfMappedValues = 0;

// determining the number of values divisible by three
long numbersDivisibleByThree = inputs.stream()
    .mapToInt(s -> {
        // variables declared outside of an anonymous function cannot be used,
        numberOfMappedValues++;
        return Integer.valueOf(s);
    }).filter(value -> value % 3 == 0)
```

Stream Methods

Stream methods can be roughly divided into two categories: (1) intermediate operations intended for processing elements and (2) terminal operations that end the processing of elements. Both of the `filter` and `mapToInt` methods shown in the previous example are intermediate operations. Intermediate operations return a value that can be further processed - you could, in practice, have an infinite number of intermediate operations chained sequentially (& separated by a dot). On the other hand, the `average` method seen in the previous example is a terminal operation. A terminal operation returns a value to be processed, which is formed from, for instance, stream elements.

The figure below illustrates how a stream works. The starting point (1) is a list with values. When the `stream()` method is called on a list, (2) a stream of list values is created. The values are then dealt with individually. The stream values can be (3) filtered by the `filter` method, which removes values that fail to meet the condition from the stream. The stream's `map` method (4) can be used to map values in a stream from one form to another. The `collect` method (5) collects the values in a stream into a collection provided to it, such as a list.



Underneath is a program of the example depicted in the image above. In this example stream, a new ArrayList list is created to which values are added. This is done in the last line `.collect(Collectors.toCollection(ArrayList::new));`.

```
List<Integer> list = new ArrayList<>();
list.add(3);
list.add(7);
list.add(4);
list.add(2);
list.add(6);

ArrayList<Integer> values = list.stream()
    .filter(value -> value > 5)
    .map(value -> value * 2)
    .collect(Collectors.toCollection(ArrayList::new));
```

Programming exercise:

Positive Numbers

Points 1/1

In the exercise template, implement the class method `public static List<Integer> positive(List<Integer> numbers)`, which receives an ArrayList of integers, and returns the positive integers from the list.

Implement the method using stream! For collecting the numbers try the command `Collectors.toList()` in addition to the `Collectors.toCollection(ArrayList::new)` command.

Exercise submission instructions

How to see the solution

Terminal Operations

Let's take a look at four terminal operations: the `count` method for counting the number of values on a list, the `forEach` method for going a through list values, the `collect` method for gathering the list values into a data structure, and the `reduce` method for combining the list items.

The `count` method informs us of the number of values in the stream as a `long`-type variable.

```
List<Integer> values = new ArrayList<>();
values.add(3);
values.add(2);
values.add(17);
values.add(6);
values.add(8);

System.out.println("Values: " + values.stream().count());
```

Sample output

```
Values: 5
```

The `forEach` method defines what is done to each list value and terminates the stream processing. In the example below, we first create a list of numbers, after which we only print the numbers that are divisible by two.

```
List<Integer> values = new ArrayList<>();
values.add(3);
values.add(2);
values.add(17);
values.add(6);
values.add(8);

values.stream()
    .filter(value -> value % 2 == 0)
    .forEach(value -> System.out.println(value));
```

Sample output

```
2
6
8
```

You can use the `collect` method to collect stream values into another collection. The example below creates a new list containing only positive values. The `collect` method is given as a parameter to the Collectors object to which the stream values are collected - for example, calling `Collectors.toCollection(ArrayList::new)` creates a new ArrayList object that holds the collected values.

```
List<Integer> values = new ArrayList<>();
values.add(3);
values.add(2);
values.add(-17);
values.add(-6);
values.add(8);

ArrayList<Integer> positives = values.stream()
    .filter(value -> value > 0)
    .collect(Collectors.toCollection(ArrayList::new));

positives.stream()
    .forEach(value -> System.out.println(value));
```

Sample output

```
3
2
8
```

Quiz:
Filter

Points:
1/1

Let's examine the following program.

```
List<Integer> numbers = new ArrayList<>();
for (int i = 0; i < 10; i++) {
    numbers.add(i);
}

long howManyNumbers = numbers.stream()
    .filter(i -> i < 4)
    .map(i -> i * 2)
    .filter(i -> i > 10)
    .count();

System.out.println("Numbers: " + howManyNumbers);
```

How many numbers does the stream take to the lambda-expression "i -> i > 10"?

Select the correct answer

- 12
- 10
- 6
- 5

✓ 4

0

The answer is correct

Answered Tries remaining: 1

Programming exercise:

Divisible

Points 1/1

The exercise template includes a template for the method `public static ArrayList<Integer> divisible(ArrayList<Integer> numbers)`. Implement a functionality there that gathers numbers divisible by two, three or five from the list it receives as a parameter, and returns them as a new list. The list received as a parameter must not be altered.

```
public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(3);
        numbers.add(2);
        numbers.add(-17);
        numbers.add(-5);
        numbers.add(7);

        ArrayList<Integer> divisible = divisible(numbers);

        divisible.stream()
                .forEach(num -> System.out.println(num));
    }
```

Sample output

```
3
2
-5
```

Exercise submission instructions

How to see the solution

The `reduce` method is useful when you want to combine stream elements to some other form. The parameters accepted by the method have the following format: `reduce(*initialState*, (*previous*, *object*) -> *actions on the object*)`.

As an example, you can calculate the sum of an integer list using the reduce method as follows.

```
ArrayList<Integer> values = new ArrayList<>();
values.add(7);
values.add(3);
values.add(2);
values.add(1);

int sum = values.stream()
    .reduce(0, (previousSum, value) -> previousSum + value);
System.out.println(sum);
```

Sample output

```
13
```

In the same way, we can form a combined row-separated string from a list of strings.

```
ArrayList<String> words = new ArrayList<>();
words.add("First");
words.add("Second");
words.add("Third");
```

```
words.add("Fourth");

String combined = words.stream()
    .reduce("", (previousString, word) -> previousString + word + "\n");
System.out.println(combined);
```

Sample output

```
First
Second
Third
Fourth
```

Intermediate Operations

Intermediate stream operations are methods that return a stream. Since the value returned is a stream, we can call intermediate operations sequentially. Typical intermediate operations include converting a value from one form to another using `map` and its more specific form `mapToInt` used for converting a stream to an integer stream. Other ones include filtering values with `filter`, identifying unique values with `distinct`, and arranging values with `sorted` (if possible).

Let's look at these methods in action through a few problems. Say we have the following Person class.

```
public class Person {
    private String firstName;
    private String lastName;
    private int birthYear;

    public Person(String firstName, String lastName, int birthYear) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.birthYear = birthYear;
    }

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
```

```
    }

    public int getBirthYear() {
        return this.birthYear;
    }
}
```

Problem 1: You'll receive a list of persons. Print the number of persons born before the year 1970.

We'll use the `filter` method for filtering through only those persons who were born before the year 1970. We then count their number using the method `count`.

```
// suppose we have a list of persons
// ArrayList<Person> persons = new ArrayList<>();

long count = persons.stream()
    .filter(person -> person.getBirthYear() < 1970)
    .count();
System.out.println("Count: " + count);
```

Problem 2: You'll receive a list of persons. How many persons' first names start with the letter "A"?

Let's use the `filter`-method to narrow down the persons to those whose first name starts with the letter "A". Afterwards, we'll calculate the number of persons with the `count`-method.

```
// suppose we have a list of persons
// ArrayList<Person> persons = new ArrayList<>();

long count = persons.stream()
    .filter(person -> person.getFirstName().startsWith("A"))
    .count();
System.out.println("Count: " + count);
```

Problem 3: You'll receive a list of persons. Print the number of unique first names in alphabetical order

First we'll use the `map` method to change a stream containing person objects into a stream containing first names. After that we'll call the `distinct`-method, that returns a stream that only contains unique values. Next, we call the method `sorted`, which sorts the strings. Finally, we call the method `forEach`, that is used to print the strings.

```
// suppose we have a list of persons
// ArrayList<Person> persons = new ArrayList<>();

persons.stream()
    .map(person -> person.getFirstName())
    .distinct()
    .sorted()
    .forEach(name -> System.out.println(name));
```

The `distinct`-method described above uses the `equals`-method that is in all objects for comparing whether two strings are the same. The `sorted`-method on the other hand is able to sort objects that contain some kind of order — examples of this kind of objects are for example numbers and strings.

Programming exercise:

Printing User Input

Points

1/1

Write a program that reads the user's input as strings. When the user inputs an empty string (only presses enter), the input reading will be stopped and the program will print all the user inputs.

Sample output

```
first
second
war is peace: 1984

first
second
war is peace: 1984
```

Exercise submission instructions

How to see the solution

Programming exercise:

Limited numbers

Points 1/1

Write a program that reads user input. When the user gives a negative number as an input, the input reading will be stopped. After this, print all the numbers the user has given as input that are between 1 and 5.

Sample output

```
7
14
4
5
4
-1
4
5
4
```

Exercise submission instructions

How to see the solution

Programming exercise:

Unique last names

Points 1/1

The exercise template contains a sketch of a program that reads user-provided information about people. Expand the program so that it will print all the unique last names of the user-provided people in alphabetical order.

Sample output

```
Continue personal information input? "quit" ends:

Input first name: Ada
Input last name: Lovelace
Input the year of birth: 1815

Continue personal information input? "quit" ends:

Input first name: Grace
Input last name: Hopper
Input the year of birth: 1906

Continue personal information input? "quit" ends:

Input first name: Alan
Input last name: Turing
Input the year of birth: 1912

Continue personal information input? "quit" ends:
quit

Unique last names in alphabetical order:
Hopper
Lovelace
Turing
```

Exercise submission instructions

How to see the solution

Objects and Stream

Handling objects using stream methods is natural. Each stream method that deals with the stream's values also enables you to call methods related to values. Let's look at an example where we have books with authors. The classes `Person` and `Book` are provided below.

```
public class Person {
    private String name;
    private int birthYear;

    public Person(String name, int birthYear) {
        this.name = name;
        this.birthYear = birthYear;
    }

    public String getName() {
        return this.name;
    }

    public int getBirthYear() {
        return this.birthYear;
    }

    public String toString() {
        return this.name + " (" + this.birthYear + ")";
    }
}
```

```
public class Book {
    private Person author;
    private String name;
    private int pages;

    public Book(Person author, String name, int pages) {
        this.author = author;
        this.name = name;
        this.pages = pages;
    }

    public Person getAuthor() {
        return this.author;
    }

    public String getName() {
```

```
        return this.name;
    }

    public int getPages() {
        return this.pages;
    }
}
```

Say we have a list of books. Calculating the average of the authors' birth years can be done using stream methods in a way that feels natural. First, we convert the stream of books to a stream of persons, and then we convert the stream of person to a stream of birth years. Finally, we ask the (integer) stream for an average.

```
// let's assume that we have a list of books
// List<Book> books = new ArrayList<>();

double average = books.stream()
    .map(book -> book.getAuthor())
    .mapToInt(author -> author.getBirthYear())
    .average()
    .getAsDouble();

System.out.println("Average of the authors' birth years: " + average);

// the mapping of a book to an author could also be done with a single map call
// double average = books.stream()
//     .mapToInt(book -> book.getAuthor().getBirthYear())
//     ...
```

Similarly, the names of the authors of books with the word "Potter" in their titles are outputted the following way.

```
// let's assume that we have a list of books
// List<Book> books = new ArrayList<>();

books.stream()
    .filter(book -> book.getName().contains("Potter"))
    .map(book -> book.getAuthor())
    .forEach(author -> System.out.println(author));
```

Streams can also be used to build more complex string representations. In the example below, we print "Author Name: Book" pairs arranged in alphabetical order.

```
// let's assume that we have a list of books at our disposal
// ArrayList<Book> books = new ArrayList<>();

books.stream()
     .map(book -> book.getAuthor().getName() + ": " + book.getName())
     .sorted()
     .forEach(name -> System.out.println(name));
```

Programming exercise:

Weighting (2 parts)

Points

7/7

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

The exercise template includes the probably familiar project "Cargo hold". However, in this exercise you need to make the for-loop using methods to use streams. The final product should not have any `while (...)` or `for (...)`-loops.

Exercise submission instructions

How to see the solution

Files and Streams

Streams are also very handy in handling files. The file is read in stream form using Java's ready-made Files class. The `lines` method in the files class allows you to create an input stream from a file, allowing you to

process the rows one by one. The `lines` method gets a path as its parameter, which is created using the `get` method in the Paths class. The `get` method is provided a string describing the file path.

The example below reads all the lines in "file.txt" and adds them to the list.

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

try {
    Files.lines(Paths.get("file.txt")).forEach(row -> rows.add(row));
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}

// do something with the read lines
```

If the file is both found and read successfully, the lines of the "file.txt" file will be in the `rows` list variable at the end of the program. However, if a file cannot be found or read, an error message will be displayed. Below is one possibility:

Sample output

```
Error: file.txt (No such file or directory)
```

Programming exercise:

Reading Files Per Line

Points 1/1

Implement the static method `public static List<String> read(String file)`, which reads the file with the filename of the parameter and returns the lines as a string list.

Exercise submission instructions

How to see the solution

Stream methods make the reading of files that are of predefined format relatively straightforward. Let's look at a scenario where a file contains some personal information. Details of each person is on their own line: first the person's name, then a semicolon, and finally the person's birth year. The file format is as follows:

Sample output

```
Kaarlo Juho Ståhlberg; 1865
Lauri Kristian Relander; 1883
Pehr Evind Svinhufvud; 1861
Kyösti Kallio; 1873
Risto Heikki Ryti; 1889
Carl Gustaf Emil Mannerheim; 1867
Juho Kusti Paasikivi; 1870
Urho Kaleva Kekkonen; 1900
Mauno Henrik Koivisto; 1923
Martti Oiva Kalevi Ahtisaari; 1937
Tarja Kaarina Halonen; 1943
Sauli Väinämö Niinistö; 1948
```

Let's assume that the file is named `presidents.txt`. Reading the details of the persons happens as follows:

```
List<Person> presidents = new ArrayList<>();
try {
    // reading the "presidents.txt" file line by line
    Files.lines(Paths.get("presidents.txt"))
        // splitting the row into parts on the ";" character
        .map(row -> row.split(";"))
        // deleting the split rows that have less than two parts (we want the r
        .filter(parts -> parts.length >= 2)
        // creating persons from the parts
        .map(parts -> new Person(parts[0], Integer.valueOf(parts[1])))
        // and finally add the persons to the list
        .forEach(person -> presidents.add(person));
} catch (Exception e) {
```

```
    System.out.println("Error: " + e.getMessage());
}

// now the presidents are on the list as person objects
```

Programming exercise:

Books from file

Points 1/1

Add the method `public static List<Book> readBooks(String file)` for the class `BooksFromFile`. It should read the file given as the parameter and forms book data from it.

The exercise template contains the class `Book`, which is used for describing a book. You should presume that the book files are in the following format:

```
name,publishing year,page count,author
```

The name and the author of the book are processed as strings, and the publishing year and the page count are processed as integers. Example of contents of a book file:

```
Do Androids Dream of Electric Sheep?,1968,210,Philip K. Dick
Love in the Time of Cholera,1985,348,Gabriel Garcia Marquez
```

Exercise submission instructions

How to see the solution

You have reached the end of this section! Continue to the next section:

→ 2. The Comparable Interface

Remember to check your points from the ball on the bottom-right corner of the material!

In this part:

Source code of the material

This course is created by the Agile Education Research -research group of the University of Helsinki.

Credits and about the material.



HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

MASSIIVISET AVOIMET VERKKOKURSSIT
MASSIVE OPEN ONLINE COURSES · MOOC.FI