The purpose of this assignment is to practice programming with lambda expressions.

Exercise 1 (acquire the expertise) – Population

Import the "es1" folder into your Integrated Development Environment (IDE). This program generates a random list of individuals (*Students*, *Workers*, and *Person*) and provides statistical information about the population. The computation of these statistics relies on anonymous inner classes and nested interfaces.
The objective of this exercise is to introduce *search()*, *categorizeString()* and *categorizeInteger()* utility methods that can accept lambda expressions as parameters. These methods should then allow to replace the repetitive 'for' loops and the anonymous inner classes found within the program.
To achieve this objective, perform the following steps:

1) Based on the logic found in the 'main' method, add the following utility methods:
   a) `List<Person> search(List<Person> population, EvaluateOperation op)`
      This method should return a new list containing instances of the *Person* class for which the operation provided as an argument evaluates to 'true'.
   b) `Map<String, List<Person>> categorizeString(List<Person> population, CategorizeOperation<String> op)`
      `Map<Integer, List<Person>> categorizeInteger(List<Person> population, CategorizeOperation<Integer> op)`
      These methods should return a map with 'String' or 'Integer' keys, respectively, and lists of persons categorized by the 'CategorizeOperation' as the corresponding values.
2) In the 'main' method, replace the existing loops with the newly created utility methods.
3) Annotate both the 'EvaluateOperation' and 'CategorizeOperation' interfaces with the @FunctionalInterface annotation. Furthermore, replace anonymous inner classes with lambda expressions when invoking the 'search,' 'categorizeString,' and 'categorizeInteger' methods.

Exercise 2 (problem solving) – Text analyzer

Write a program that reads a text file and outputs the following information:

- the longest word
- the shortest word with at least 5 characters
- the word that contains the highest number of vowels
- the list of words that start with a vowel
- the list of words that start with a capital T

Create two auxiliary functions, 'find' and 'findAll,' which are used for collecting the required information, along with the lambda expressions passed as arguments, providing the specific logic of the search. The methods must have the following signature:

- `private static String find(List<String> words, BiPredicate<String, String> operation)`
- `private static List<String> findAll(List<String> words, Predicate<String> operation)`

HINTS:

- You can read the contents of a text file into a list of words using the following one-liner:
  ```
  Arrays.asList(new String(Files.readAllBytes(Paths.get("/path/to/file.txt"))).split("\\s+"))
  ```
- The `BiPredicate<String, String> operation` parameter of the *find* method is used to compare two strings.
  For instance, when searching for the longest word, the first parameter could be the longest word found so far, and the second parameter is the current word being tested. If the expression returns true, a new longest word is discovered.

Exercise 3 (optional deepening) – Sorting

Open the *es3/SortingExample* file in your IDE. The program creates a list of random integers, sorts them by calling the *list.sort(new MySortingLogic())* method and then prints the sorted list to the console.

The objective of this exercise is to implement the same sorting logic using the following alternatives instead of creating a new instance of the *MySortingLogic* class when calling *list.sort()*:

- an anonymous inner class,
- a lambda expression,
- an instance method reference of the *MySortingLogic* class,
- a static method reference, by adding a static method to the *SortingExample* class.

Develop these alternatives in their respective methods (*testAnonymousInnerClass, testLambda, testInstanceMethodReference* and *testStaticMethodReference*).