The purpose of this assignment is to practice the development of Java programs with nested classes and interfaces, including local and anonymous classes.

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 helper classes and interfaces found in the "operations" package.
Your objective is to encapsulate the implementation of these helper classes and interfaces by eliminating the "operations" package and refactoring all its contained interfaces and classes as nested interfaces/classes within the "S3Es1" class.
Please execute this task in three distinct versions:

1. Utilize nested static classes and nested interfaces within the "S3Es1" class.
2. Employ nested local classes within the main method.
3. Make use of anonymous classes within the main method.

Note: Classes within the "person" package should remain unaltered, and the primary focus should be on removing the "operations" package.

Exercise 2 (problem solving) – Advanced stack

Import the *AdvancedStack* interface in your Integrated Development Environment (IDE). This interface contains the standard stack operations like *push()*, *pop()*, *peek()*, *size()*, and *isEmpty()*. Additionally, it introduces a *count()* method, which accepts as parameter any object implementing the *AdvancedStack.Evaluate* interface. The *count()* method's role is to verify each element within the stack and count the number of elements that return true when tested with the provided *AdvancedStack.Evaluate* object (this is achieved by invoking the *AdvancedStack.Evaluate.verify()* method with a stack element's value instance as a parameter).
Create a *CustomStack* class that implements the *AdvancedStack* interface. You are not allowed to use arrays or *Collections* like *ArrayList* or *LinkedList*. Instead, introduce a private inner class called *StackElement* within the *CustomStack* class to serve as a list node for storing pushed values. It's important to note that the *StackElement* class should not be accessible from the client code that uses the *CustomStack* class, as its purpose is strictly internal to the stack's implementation logic.
To validate your implementation, push a few objects (of type *String*, *Integer*, *Float*, ...) to the stack and verify that the *count()* method returns the correct number of items by creating at least one local class implementation and one anonymous class of the *AdvancedStack.Evaluate* interface.

Note: some possible *AdvancedStack.Evaluate* specialization classes could test:

- That the object is an instance of the *Integer* class
- That the object is a *String* containing "test"
- That the object is a *Float* or *Double* and its value is greater than 0.5
- That the object represents an even number

Exercise 3 (optional deepening) – Sorting

Starting from the provided *SortingDemo* class, which serves as a template program for testing different sorting algorithms, your task is to implement three sorting algorithms by creating distinct classes that implement the *Comparator* interface. These three sorting algorithms are to be applied to the given sample input data [5, 2, 9, 1, 5, 6, 3, 8], and they should produce the following results:

1. **Ascending Order**: this should be implemented as a nested class (e.g., AscendingComparator). The result should be [1, 2, 3, 5, 5, 6, 8, 9].

2. **Descending Order**: this should be implemented as a local class (e.g., DescendingComparator). The result should be [9, 8, 6, 5, 5, 3, 2, 1].

3. **Ascending Order with Odd Numbers First**: this should be implemented as an anonymous inner class. The result should be [2, 6, 8, 1, 3, 5, 5, 9].