Assignment 1 (1 point) - Deadline Aug 22 23:59 hrs

Learning goals

This assignment is about unit testing and basic fork-join parallelism in Java.
You are not allowed to use any class from `java.util.concurrent` for this assignment.
You are expected to learn the following skills by completing this assignment:

- How to start new threads in Java.
- How to wait for threads to finish and read a result back.
- Writing and running unit tests using JUnit.

1. Mutations and combinations

First a couple of definitions we will use for this assignment: A *mutation* defines a method that changes the state of a given object. A *combination* defines an associative binary method that combines two objects into one. It also defines a neutral object that is the identity element of the combine operations (i.e. any object combined with the neutral object becomes itself). We will consider a couple of specific mutation for *employees* and combinations for integers. Read the supplied interfaces `assignment1.Mutator` and `assignment1.Combinator` and the supplied class `assignment1.Employee`.

a. Create the following classes, each of them implementing the interface `Mutator<Employee>`:

   1. `IncreaseSalary`: If the employee is older than 40 years, increase his salary by half his age.
   2. `LowerCaseName`: Change the name of the employee such that all letters are lower case.

b. Create the following classes:

   1. `AddSalary implements Combinator<Employee, Integer>`: `get` gets the salary of an employee. `combine` adds two salaries.
   2. `LongestName implements Combinator<Employee, String>`: `get` gets the name of an employee. `combine` combines two names by simply returning the longest.

c. Test the combination implementations using JUnit.

   1. Create a new class called `CombinatorTest`.
   2. Write a private method called `genEmployee` that returns a new employee with the name "John Doe", an age between 20 and 60 and a salary between 3000 and 5000. The age and salary should be selected from a uniform pseudo-random distribution using `java.util.Random`.

3. Write a JUnit test verifying both associativity[1] and neutral element[2] for both `AddSalary` and `LongestName`. The test should generate at least a thousand test cases using `genEmployee` and check the invariants.

2. List mutations

A *list mutation* is a generalization of a mutation to a list of objects. Given a mutation and a list, a list mutation applies the mutation to every element of the list. Read the supplied interface `assignment1.ListMutator`.

a. Write a JUnit test that verifies a map of `IncreaseSalary`.

   1. Create a new class called `ListMutatorTest`.
   2. Write a private method called `dataSet` that returns a fixed list of 10 employees where you select the names, ages and salaries. Make sure that at least 10% are older than 40.
   3. Without using Java, calculate the expected sum of the salaries after a map of `IncreaseSalary` on your data set.
   4. Write a private method called `testListMutation` that takes a `ListMutator` implementation as input, runs a map of `IncreaseSalary` on your data set, sums the salaries of the employees and asserts the equality of the actual sum of salaries with the expected.

b. Implement a sequential list mutation:

   1. `ListMutatorSequential`: Use just the main thread (i.e. using a simple for-loop).
   2. Check `ListMutatorSequential` by writing a JUnit test that uses the `testListMutation` method.

c. Implement a parallel list mutation:

   1. `ListMutatorParallel`: Start one thread per element in the list.
   2. Check `ListMutatorParallel` by writing a JUnit test that uses the `testListMutation` method.

d. Implement a piece-wise parallel list mutation:

   1. `ListMutatorChunked`: Implement the list mutation using no more than 3 active threads at any given time (besides the main thread). Each thread should do approximately the same amount of work.

      You are allowed to use your implementations of `ListMutatorSequential` and `ListMutatorParallel`.
   2. Check `ListMutatorChunked` by writing a JUnit test that uses the `testListMutation` method.

[1] $combine(x, combine(y, z)) = combine(combine(x, y), z)$
[2] $combine(neutral(), x) = x = combine(x, neutral())$

3. Aggregators

An *aggregator* computes a single value from a list of objects. It does so by using a supplied *combination*. `get(T x)` is called on each element of the list, and the get'ed values are successively combined into a single S using `combine(S x, S y)`. Read the supplied interface `assignment1.Aggregator`.

a. Write a JUnit test that verifies an aggregation of `AddSalary`.

   1. Create a new class called `AggregatorTest`.
   2. Write a private method called `dataSet` that returns a fixed list of 10 employees where you select the names, ages and salaries. Make sure that all salaries are positive non-zero numbers.
   3. Without using Java, calculate the sum of the salaries.
   4. Write a private method called `testAggregation` that takes an `Aggregator` implementation as input, runs an aggregation of `AddSalary` and asserts the equality of the actual sum of salaries with the expected.

b. Implement a sequential aggregation:

   1. `AggregatorSequential`: Implement aggregation using no other threads than the main thread (i.e. using a simple for-loop).
   2. Check `AggregatorSequential` by writing a JUnit test that uses the `testAggregation` method.

c. Implement a parallel aggregation:

   1. `AggregatorParallel`: Implement a divide-and-conquer parallel aggregation. Each thread should either compute one instance of `combine(S x, S y)` or return a trivial result.
   2. Check `AggregatorParallel` by writing a JUnit test that uses the `testAggregation` method.