# Assignment 1 - Deadline Nov 10 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

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). For this assignment we will consider a couple of specific mutation for *employees* and combinations for integers. Read the supplied interfaces `assignment1.Mutation` and `assignment1.Combination` and the supplied class `assignment1.Employee`.

a. Implement the following mutations on employees:

   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. Implement the following combinations:

   1. `AddSalary`: Addition of salaries.
   2. `MinAge`: Minimum of ages.

c. Test the combination implementations using JUnit.

   1. Create a new class called `CombinationTest`.
   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 `MinAge`. The test should generate at least a thousand test cases using `genEmployee` and check the invariants.

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

## 2. Maps

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

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

   1. Create a new class called `MapTest`.
   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 `testMap` that takes a `Map` 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 map:

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

c. Implement a parallel map:

   1. `MapParallel`: Implement map by starting one thread per mutation.
   2. Check `MapParallel` by writing a JUnit test that uses the `testMap` method.

d. Implement a piece-wise parallel map:

   1. `MapChunked`: Implement map as a parallel map, but having 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 `MapSequential` and `MapParallel`.
   2. Check `MapChunked` by writing a JUnit test that uses the `testMap` method.

## 3. Aggregations

An *aggregation* computes a single value given a list of objects, by combining all the values of the list using a given combination. Read the supplied interface `assignment1.Aggregation`.

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

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

c. Implement a parallel aggregation:

   1. `AggregationParallel`: Implement aggregation by starting one thread per call to the `combine` method.
   2. Check `AggregationParallel` by writing a JUnit test that uses the `testAggregation` method.