# Exercises week 11

Last update 2024/11/10

These exercises aim to give you practical experience with network computing using the the internet protocols TCP (Java sockets) and HTTP (HttpURlConnection).

## Reading material

- Oracle's documentation of Java Sockets
- A simple tutorial Source code for a simple example of socket communication.
- Oracle's documentation of HttpURlConnection.

**Exercise 11.1** In this exercise you will try running an `EchoServer` and `EchoClient` on your own PC.

*Mandatory*

1. Compile and run the code for the tutorial `EchoServer` and `EchoClient`. **You need to open two command line windows (shells): one for running the server and one for running the client!!!!**
2. Modify the code for the `EchoClient` and `EchoServer` so any word sent by the client is capitalized by the server and then returned. Test your revised code on a few examples to demonstrate that the changed code work as expected.

**Exercise 11.2** In this exercise you must perform, on your own hardware, the measurement performed in the lecture using the example code in the files `NumberServer.java` and `SocketCountingThreads.java`.

*Mandatory*

1. In the run method of `SocketCountingThreads.java` there are three lines, two of them are inactive (because they are commented out). The first experiment is running the "noLocking" version.

   Compile and run the code files `NumberServer` and `SocketCountingThreads`. As in the first exercise above you need two command line windows. In this version there can be race conditions in the server because the two commands get and put are not executed as one atomic operation.

   Before closing the NumberServer prints the value of its counter. Record this number and inspects the code in `SocketCountingThreads`. What would be the result if there where no race conditions?

   Record the execution time (measured with Mark7) reported when running `SocketCountingThreads`.

2. Repeat the timing experiment by moving the comments in the run method, so you measure the excution times for `serverLocking()` and `clientLocking()`.

   Collect all the results of the three timing experiments in a table and comment on the results. Are they as expected? Why/why not?

   *Challenging*

3. If you have two PC's set up an experiment where the server is running on one PC and `SocketCountingThreads` on the other. Change the value of the string `URL` (top of the source code) to the IP address of the server PC.

   Repeat the timing experiments for `serverLocking()`, `clientLocking()` and compare the results with the measurements where the client and server ran on the same PC.

**Exercise 11.3 Not mandatory**

1. Try to run the code in BusDepart.java.
2. Find the code for a train station/bus stop near your home (see slides lecture11.pdf). Replace the code for ITU in BusDepart.java with the code of your bus stop. Compile and run your "personalized" rejseplan.

## Basic functional programming exercises in Erlang

The goal of these exercises is to practice basic functional programming concepts in Erlang.

**Exercise 11.4** *This exercise is optional and unlabeled. You do not need to do this exercise. However, if you are not used to working functional programming in Erlang, this exercise will get you started. We will not cover this exercises at the feedback session unless you ask about it.*

1. Create a module called `functional_erlang`. You will implement all the exercises below within this module. If you would like to include this exercise to your submission, use a root folder `assignment5`, which contains two directories: `week11exercises` (with the Gradle project for the mandatory exercises) and `week11Erlang` with the module for this exercise.
2. Implement a function `remove(List, Elem)`, which takes a list and an element and returns a new list with all occurrences of `Elem` removed. Implement the function with and without using list comprehensions.
3. Implement a function `count(List, Elem)`, which returns the number of repetitions of `Elem` in the `List`.
4. Implement a function `count_occurrences(List, Pred)`, which returns the number of elements in `List` that satisfy a predicate `Pred`. The predicate `Pred` must be implement as a function that given an element of `List` returns a Boolean value indicating whether the predicate holds for the element. Implement the function with and without using list comprehensions.
5. Implement a function `filter(List, Pred)`, which returns a new list with only the elements of `List` that satisfy `Pred`. As before, the predicate `Pred` is defined as a function returning a Boolean value for an element in `List`. Implement the function with and without using list comprehensions.
6. Implement a function `flatten(Lists)` that returns a single list containing all the elements of a list of lists `Lists`.
7. Implement a function `fold(List, Fun, Acc)` that iterates `List` with elements $e_1, e_2, \ldots$ from left to right and applies function `Fun`—which takes two parameters and returns a value of the same type than the elements—and reduces the `List` to a single element. In each iteration, the `fold` function applies `Fun` to the result of applying `Fun` to the all previous elements in the list (the accumulated result). In the first iteration, the accumulated result is `Acc`. For instance, given the list `[1,2,3]`, function `fun (X,Y) X+Y end`, and `Acc=0`, the function `fold()` computes first `0+1=1`, then it uses the result of this computation to execute `1+2=3` and finally `3+3=6`.
8. Implement a function `what_is_the_temperature(Temperature, Scale, City)` that prints in standard input: "It is `Temperature` degrees `Scale` in `City`"—of course, replacing the variables in the previous sentence with the corresponding values from the input parameters.
9. Implement a satisfaction checking function for propositional logic. This exercise requires a few preliminary steps:
   (a) Use records to encode propositional formulae in Erlang. To keep it simple, we consider formulae that can be composed of propositions such as $p, q, \ldots$, and the logical connectives $\wedge$ and $\neg$; note that this is without loss of generality as the remaining connectives can be defined in terms of conjunction and negation.
   (b) A model in propositional logic is map from propositions to truth values (true or false). For simplicity, implement models as sets containing the propositions that are true in the model. Thus, any proposition appearing in the formula that are not in the set are interpreted as being false in the model.

   Given the above, the function should traverse the structure of the formula and decide its truth value. For instance, given the formula $p \wedge q$ and the model $[p \mapsto \mathit{true}, q \mapsto \mathit{false}]$, your function should return false, and for the formulae $p \wedge \neg q$ the same model should return `true`.