Exercises week 1

Last update: 2025/08/23

Practical info before you start

Abbreviations

The following abbreviations are used in the exercise sheets:

- "Goetz" means Goetz *et al.*, *Java Concurrency in Practice*, Addison-Wesley 2006.
- "Herlihy" means Herlihy *et al.*, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2020.
- "ConcNote" means *Concurrency I + II by Raúl Pardo and Jørgen Staunstrup*

The exercises are a way for you to get a practical understanding of the material. In addition, they serve as the outset for bi-weekly feedback.

Oral Feedback

Feedback is in oral feedback sessions. Please see the document in our Github repository (https://github.itu.dk/jst/PCPP2025-public/blob/main/general-info/assignment-submissions-and-oral-feedback.md) regarding how to book an oral feedback session, and how feedback sessions are conducted.

Mandatory/Challenging Exercises

Exercises are labelled as *mandatory* or *challenging*. For a submission to be considered accepted, all mandatory exercises must be successfully completed (and approved). Furthermore, the code of mandatory exercises, shown during oral sessions, must compile and run. Code that is close to correct, but that does not compile and run will not be accepted; note again that this is only for mandatory assignments.

We acknowledge that different students strive to strike different balances between the different classes and between study and other aspects of their lives. Consequently, we include *challenging* exercises. These exercises are *optional*, they are not required for acceptance. However, completing the challenging exercises increases your chances of obtaining high marks at the exam.

Groups should be composed by students at the same ambition level.

Installing JDK/Gradle and Running Exercise Code

In this course, we will use the Gradle tool and Java JDK 8 or higher up to version 17 for the exercises (and lectures). We have provided a guide on how to set up the programming environment we will use int he course. The guide is in our Github repository (https://github.itu.dk/jst/PCPP2025-public/blob/main/general-info/guide-using-gradle-for-exercises.md). Please follow the guide to ensure that you set up the programming environment in your machine. The guide provides information for different operating systems. If you have any trouble on getting things running please contact us, e.g., by attending exercise sessions or posting in the Questions and Answer Forum in LearnIT.

Information about the hardware of your PC

For some exercises you need to know some details about the PC (in particular the number of cores) you use for answering the exercise. If you do not already know these details, here are some hints to get this info.

- linux use the `nproc` command
- windows Open task manager (Ctrl + Shift + Esc) and select the Performance tab
- Hit Command+Spacebar to open Spotlight, then type System Information and hit return (or type System Profiler for earlier MacOS versions) Click Hardware to see the Hardware Overview

Some of you may use a virtual machine e.g. VirtualBox or Windows Subsystem for Linux (WSL). This may impact performance. For every day use, you may not notice any performance impact. But for some the exercises in PCPP, you may see a significant impact.

Exercise 1.1 Consider the code in the file `TestLongCounterExperiments.java`. Note that this file contains a class, `LongCounter`, used by two threads:

```
class LongCounter {
  private long count = 0;
  public void increment() {
    count = count + 1;
  }
  public long get() {
    return count;
  }
}
```

Mandatory

1. The `main` method creates a LongCounter object. Then it creates and starts two threads that run concurrently, and each increments the `count` field 10 million times by calling method `increment`.

   What output values do you get? Do you get the expected output, i.e., 20 million?

2. Reduce the `counts` value from 10 million to 100, recompile, and rerun the code. It is now likely that you get the expected result (200) in every run.

   Explain how this could be. Is it guaranteed that the output is always 200?

3. The `increment` method in LongCounter uses the assignment

   ```
   count = count + 1;
   ```

   to add one to `count`. This could be expressed also as `count += 1` or as `count++`.

   Do you think it would make any difference to use one of these forms instead? Why? Change the code and run it. Do you see any difference in the results for any of these alternatives?

4. Set the value of `counts` back to 10 million. Use Java `ReentrantLock` to ensure that the output of the program equals 20 million. Explain why your solution is correct, and why no other output is possible.

   Note: In your explanation, please use the concepts and vocabulary introduced during the lecture, e.g., critical sections, interleavings, race conditions, mutual exclusion, etc.

   Note II: The notes above applies to all exercises asking you to explain the correctness of your solution.

5. By using the `ReetrantLock` in the exercise above, you have defined a *critical section*. Does your critical section contain the least number of lines of code? If so, explain why. If not, fix it and explain why your new critical section contains the least number of lines of code.

   Hint: Recall that the critical section should only include the parts of the program that only one thread can execute at the same time.

Challenging

6. Decompile the methods increment from above to see the byte code in the three versions (as is, `+=`, `++`). The basic decompiler is `javap`. The decompiler takes as input a (target) `.class` file. In Gradle projects, `.class` files are located in the directory `app/build/classes/`—after compiling the `.java` files. The flag `-c` decompiles the code of a class. Does the output of `javap` verify or refuse the explanation you provided in part 3.?

7. Extend the `LongCounter` class with a `decrement()` method which subtracts 1 from the `count` field without using locks. Change the code in `main` so that `t1` calls `decrement` 10 million times, and `t2` calls `increment` 10 million times, on a `LongCounter` instance. What should the expected output be after both threads have completed?

   Use `ReentrantLock` to ensure that the program outputs 0. Explain why your solution is correct, and why no other output is possible.

8. Again for the code in part 4 (i.e., without `decrement()`), remove the `ReetrantLock` you added. Set the variable `counts` to 3.

   What is the minimum value that the program prints? Does the minimum value change if we set `counts` to a larger number (e.g., 20 million)?

   Provide an interleaving showing how the minimum value output can occur.

Exercise 1.2

Consider this class, whose `print` method prints a dash “-”, waits for 50 milliseconds, and then prints a vertical bar “|”:

```
class Printer {
  public void print() {
    System.out.print("-");
    try { Thread.sleep(50); } catch (InterruptedException exn) { }
    System.out.print("|");
  }
}
```

Mandatory

1. Write a program that creates a Printer object `p`, and then creates and starts two threads. Each thread must call `p.print()` forever. Note: You can easily run your program using the gradle project for the exercises by placing your code in the directory `week01exercises/app/src/main/java/exercises01/` (remember to add `package exercises01;` as the first line of your Java files).

   You will observe that, most of the time, your program print the dash and bar symbols alternate neatly as in `-|-|-|-|-|-|-|`. But occasionally two bars are printed in a row, or two dashes are printed in a row, creating small “weaving faults” like those shown below:

```
--|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-||--|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-||--|-|-|-|-|-|-|-|-|-|
```

2. Describe and provide an interleaving where this happens.

3. Use Java `ReentrantLock` to ensure that the program outputs the expected sequence `-|-|-|-|....`

   Compile and run the improved program to see whether it works. Explain why your solution is correct, and why it is not possible for incorrect patterns, such as in the output above, to appear.

Challenging

4. Consider a concurrent program composed of two threads that execute the method `print()` exactly 4 times each without using locks. Is it possible that the program prints this output below?

   `--|-|-|-|-|-|-||`

   If your answer is yes, provide an interleaving that provides the required output. If your answer is no, argue why such an interleaving is not possible.

Exercise 1.3 Imagine that due to the COVID-19 pandemic Tivoli decides to limit the number of visitors to 15000. To this end, you are asked to modify the code for the turnstiles we saw in the lecture. The file `CounterThreads2Covid.java` includes a new constant `MAX_PEOPLE_COVID` equal to 15000. However, the two threads simulate 20000 people entering to the park, so unfortunately some people will not get in `:'(`.

Mandatory

1. Modify the behaviour of the `Turnstile` thread class so that that exactly 15000 enter the park; no less no more. To this end, simply add a check before incrementing `counter` to make sure that there are less than 15000 people in the park. Note that the class does not use any type of locks. You must use `ReentrantLock` to ensure that the program outputs the correct value, 15000.
2. Explain why your solution is correct, and why it always output 15000.

Exercise 1.4 In *Goetz* chapter 1.1, three motivations for concurrency is given: resource utilization, fairness and convenience. In the concurrency note there is an alternative characterization of the different motivations for concurrency (coined by the Norwegian computer scientist Kristen Nygaard):

- **Inherent** User interfaces and other kinds of input/output.
- **Exploitation** Hardware capable of simultaneously executing multiple streams of statements. A special (but important) case is communication and coordination of independent computers on the internet,
- **Hidden** Enabling several programs to share some resources in a manner where each can act as if they had sole ownership.

Mandatory

Neither of the definitions is very precise, so for this exercise, there are many possible and acceptable answers.

1. Compare the categories in the concurrency note (https://github.itu.dk/jst/PCPP2025-Public/blob/main/week01/concurrency-note/concurrencyPCPP.pdf and Goetz, try to find some examples of systems which are included in the categories of Goetz, but not in those in the concurrency note, and vice versa (if possible - if not possible, argue why).
2. Find examples of 3 systems in each of the categories in the Concurrency note which you have used yourself (as a programmer or user).

Exercise 1.5 In this exercise you must report some of the hardware features of your PC and do a simple runtime measurement.

Mandatory

1. Which operating system are you using. Is is emulated on top of another operating system?
2. Collect this information about the hardware of the PC that you will be using for the PCPP exercises: Operating system, no of cores, size of main memory, cache architecture (including sizes of the caches).
3. Java gives you access to the wall-clock time of your PC: `System.nanoTime()`. By calling it twice before and after some code, you get information about how long it took to execute the code:

   ```
   private long start, spent;
   start= System.nanoTime();
   "code you are measuring"
   spent += System.nanoTime()-start;
   ```

   Measure how long it takes to add the numbers 1, 2, ... 100 on your PC.

Basic functional programming exercises in Erlang

The goal of these exercises is to practice basic functional programming concepts in Erlang. We will not use Erlang until the end of the course. However, if your functional programming skills are rusty or you have never programmed in Erlang, it is a good idea to get started as soon as possible. There is a guide in the course repository describing how to get started with Erlang https://github.itu.dk/jst/PCPP2025-Public/blob/main/general-info/guide-using-erlang-for-exercises.md. We recommend that you start looking into Erlang from the beginning of the course, so that when we start using Erlang you are already familiar with the syntax and basic functionality. The textbook "Learn You Some Erlang for great good!" https://learnyousomeerlang.com/content is a great resource. We recommend going through the chapters from "Introduction" to "A Short Visit to Common Data Structures" to become fluent in Erlang.

Exercise 1.6 *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 `Exercise1`, which contains two directories: `week01exercises` (with the Gradle project for the mandatory/challenging exercises) and `week01Erlang` 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`.