**Lab Session #4**

**Introduction**

This lab is about experimenting with Java 8 *Streams*. As always, if you did not finish everything from the last lab, you have to make sure you catch up with any remaining tasks. In case you are confused about any of the tasks, remember we have a Moodle Discussion forum for this course, where you can ask question or post any observations.

**Task #1: Streams**

The source code for Chapter 4 of the textbook contains a number of example programs. Load them into your IDE and work through the following sub-tasks:

1. Run (and understand) the code in StreamBasic.java to see how to move from Java 7 *Collections* to Java 8 *Streams.*
2. Experiment with StreamVsCollection.java to understand the limitations when processing streams.
3. Load the file Dish.java (which has the Dish class as well as the menu data) and experiment with the code.
4. Write some simple stream filters and examine the output. For example, use streams to filter the first two meat dishes, calling it like this:

List<Dish> firstTwoMeatDishes = menu.stream()  
 .???

**Task #2: Mapping**

Here are some programming exercises using map and flatMap:

1. Given a list of numbers, return a list of the square of each number: For example, given [1, 2, 3, 4, 5] you should return [1, 4, 9, 16, 25]. Start with this template:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
List<Integer> squares = numbers.stream()  
 .???

1. Given two lists of numbers, return all pairs of numbers: For example, given a list [1, 2, 3] and a list [3, 4] you should return [(1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)]. For simplicity, you can represent a pair as an array with two elements.
2. Extend the previous example to return only pairs whose sum is divisible by 3: For example, (2, 4) and (3, 3) are valid.

**Task #3: Reducing**

The source code for Chapter 5 of the textbook contains the files Trader.java and Transaction.java. The main program PuttingIntoPractice.javacontains the solutions to the tasks shown on the slides:

1. Find all transactions in the year 2011 and sort them by value (small to high).
2. What are all the unique cities where the traders work?
3. Find all traders from Cambridge and sort them by name.
4. Return a string of all traders’ names sorted alphabetically.
5. Are any traders based in Milan?
6. Print all transactions’ values from the traders living in Cambridge.
7. What's the highest value of all the transactions?
8. Find the transaction with the smallest value.

Try to write the stream operations without looking at the solutions first and then compare your code against the solutions. Once you ran and understood all the implemented tasks, work on these additional two:

1. For each trader, return the number of *lowercase* letters in the name (hint: look at the chars method on String).
2. Find the city String with the largest number of lowercase letters from all the cities in the transaction list. Experiment with returning an Optional<String> to account for the case of an empty input list.

**Task #4: Generating Streams from Functions**

The [Fibonacci series](https://en.wikipedia.org/wiki/Fibonacci_number) is famous as a classic programming exercise. The numbers in the following sequence are part of the Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.... The first two numbers of the series are 0 and 1, and each subsequent number is the sum of the previous two.

The series of Fibonacci tuples is similar; you have a sequence of a number and its successor in the series: (0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21)....

Your task is to generate the first 20 elements of the series of Fibonacci tuples using the iterate method!

Some help to get you started: The first problem is that the iterate method takes a UnaryOperator<T> as argument and you need a stream of tuples such as (0, 1). You can, again rather sloppily, use an array of two elements to represent a tuple. For example, new int[]{0, 1} represents the first element of the Fibonacci series (0, 1). This will be the initial value of the iterate method:

Stream.iterate(new int[]{0, 1}, ???)  
 .limit(20)  
 .forEach(t -> System.out.println("(" + t[0] + "," + t[1] +")"));

In this task, you need to figure out the highlighted code with the ???. Remember that iterate will apply the given lambda successively.

**Task 5: Pythagorean triples**

The source code for Chapter 5 also contains the file NumericStreams.java that generates [Pythagorean triples](https://en.wikipedia.org/wiki/Pythagorean_triple). Read Chapter 5.6.3 to understand how it works. Now write a JUnit test to verify that the generated triples are actually correct. Compute your test coverage using JaCoCo.

That's all for this week!