Learning goals

Before the next day, you should have achieved the following learning goals:

- Launching different thread in a program
- Having a general understanding of how a non-deterministic program works.
- Synchronising access to shared resources

1 Text loops

Look at the following code (comments ommitted for brevity). What will the output be for each of the "thread" and the "no threads" modes?

```
public class TextLoop implements Runnable {
    public static final int COUNT = 10;

    private final String name;

    public TextLoop(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        for (int i = 0; i < COUNT; i++) {
            System.out.println("Loop:" + name + ", iteration:" + i + ".");
        }
    }

    public static void main(String args[]) {
        if (args.length < 1 || (!args[0].equals("0") && !args[0].equals("1"))) {
            System.out.println("USAGE: java TextLoop <mode>");
            System.out.println("     mode 0: without threads");
            System.out.println("     mode 1: with threads");
        } else if (args[0].equals("0")) {
            for (int i = 0; i < 10; i++) {
                Runnable r = new TextLoop("Thread " + i);
                r.run();
            }
        } else {
            for (int i = 0; i < 10; i++) {
                Runnable r = new TextLoop("Thread " + i);
                Thread t = new Thread(r);
                t.start();
            }
        }
    }
}
```

Compile and execute this code several times. Do you get the result you expected? Run it several times. Do you always get the same result in 'mode 0'? And in mode 1?

2 Counting

Have a look at the following code. What will be the value of the counter at the end of its execution (comments ommitted for brevity)?

```
public class Increaser implements Runnable {
   private Counter c;

   public Increaser(Counter counter) {
        this.c = counter;
   }

   public static void main(String args[]) {
      Counter counter = new Counter();
      for (int i = 0; i < 100; i++) {
         Increaser increaserTask = new Increaser(counter);
         Thread t = new Thread(increaserTask);
         t.start();
      }
   }

   public void run() {
      System.out.println("Starting at " + c.getCount());
      for (int i = 0; i < 1000; i++) {
         c.increase();
      }
      System.out.println("Stopping at " + c.getCount());
   }
}

public class Counter {
   private int n = 0;
   public void increase() {
      n++;
   }
   public int getCount() {
      return n;
   }
}
```

Compile and execute this code several times. Do you get the result you expected? Do you get always the same result? What would you change to make sure the last value of the counter is what it should be?

3 Bank account

Look at the following code of a simplified bank account (comments ommitted for brevity). Assume that there are many threads accessing this object and think what is the minimum number of `synchronized` statements that are needed to ensure a correct behaviour.

```
public class BankAccount {
    private int balance = 0;
    public int getBalance() {
        return balance;
    }
    public void deposit(int money) {
        balance = balance + money;
    }
    public int retrieve(int money) {
        int result = 0;
        if (balance > money) {
             result = money;
        } else {
```

```
                result = balance;
            }
            balance = balance - result;
            return result;
        }
    }
```

Check your answer with a colleage or with one of the members of the faculty.

4 Responsive UI

Write a program that asks from the user the length in milliseconds of ten tasks. As the user enters the length, the tasks start running in the background while the user enters the length of the other tasks. When the tasks end, the program must register it and say it unless it is waiting for the user to enter data. See this sample output:

```
Enter the duration (in ms) of task 0: 10
Enter the duration (in ms) of task 1: 3000
Finished tasks: 0
Enter the duration (in ms) of task 2: 2000
Enter the duration (in ms) of task 3: 1000
Enter the duration (in ms) of task 4: 10
Enter the duration (in ms) of task 5: 1000
Finished tasks: 2, 1, 3, 4
Enter the duration (in ms) of task 6:
...
```

Note that several tasks may end in between two user inputs.

5 Parallel computation

Look at the attached program `ComputationLauncher` for an example of a heavy computation being performed in sequence or in parallel using more than one processor at the same time. In an old machine with two processors, the output looks like:

```
Result: 7.999582837247596E14
Time without threads: 11110ms
Result: 7.999582837247596E14
Time with threads: 6326ms
```

Make sure you understand how the program works. How would you modify the program if you machine had four processors? You can see how many processors (or cores) your machine has by reading the value of:

```
Runtime.getRuntime().availableProcessors();
```

6 Immutability (*)

Look at the attached program `ImmutableExample`. Read it carefully. Do you see any flaws? If yes, what whould you change to make the program work without problems?

What whould you change to make the IDCard class immutable?

7 Self-ordering list (*)

One of the advantages of using threads in some applications is that they allow programmers to do costly operations in the background when nobody is looking. For example, a self-sorting list can reorder itself in between calls to minimise the time needed to add elements (e.g. in applications where elements are added in big batches, but consulted only rarely).

Hint: For this exercise, it may be easier to create your own dynamic list instead of relying on those from the Java Collections Library. Remember that the latter are not thread-safe, i.e. they are not properly synchronised, so they require external synchronisation.

7.1 a)

Create a list of Integer that keeps itself sorted by sorting itself when others are not looking. The list must have at least methods `add(Integer)` (to add new Integers) and `get(int)` (to get the i^{th} integer in the (sorted) list).

- The `add()` method must add the new element at the end of the list, mark the list as "not sorted" and return immediately.
- Another thread must be always observing the list and checking whether it is sorted; if it is, it must mark it as "sorted"; if it is not, the thread must sort it. The thread must re-order the list in small steps to make sure it does not cause additional delay when new elements are added to the list while it is sorting itself (i.e. do not include a long loop in your synchronised code that may block the addition of new elements for a long time).
- The method `get(int)` must always return the i^{th} integer in the (sorted) list; if this method is called while the list is "not sorted", the list must be fully sorted before it can return; no additional elements can be added until the call of `get(int)` returns.

7.2 b)

Create another self-ordering list in which the `add()` method launches a new thread whose only purpose is to add the new integer at the right position. (Hint: if the new element has to be placed at the beginning of the list, there is no need to launch a thread only for that).

8 Dining philosophers (*)

A group of weak-sighted philosophers gathered around a circular table to have dinner together. They were served on one big round plate in the middle of the table, and they were given a fork each (so there was a fork on the table between every two adjacent philosophers). In order to prevent everybody to take food at the same time, risking their clumsy hands hacking at their neighbours' with the forks, the philosophers discussed and agreed to adhere to the following protocol: each philosopher could only get food if it had grabbed both forks on their left and their right.

Write a program that implements a dinner with n philosophers and n forks. Make sure that each fork can be grabbed by only one philosopher at a time. Verify that a naive synchronisation strategy can lead very quickly to a deadlock (when/why does this happen?); fix the program so that this cannot happen.

(Hint: during development, it may be helpful to implement a monitoring class (waiter?) that checks what the philosophers are doing, e.g. how many forks they have grabbed).