Advanced Object Oriented Programming, DT4014, VT23
Assignment 1 (due **March 31**)

Wojciech Mostowski & Nesma Rezk
Halmstad University, ITE

Introduction

You are expected to work as a team in the group you were assigned. Please submit your asnwers through the Blackboard submission system. Include the ZIP file with your code and the short PDF report that summarises your work and answers any "textual" questions that were asked.

The exercises are taken from chapters 2, 3 & 4 in the course book. The site

http://bcs.wiley.com/he-bcs/Books?action=index&itemId=0471744875&bcsId=2561

has solutions to selected exercises. You have to choose **Browse by chapter** and check the **Student Solutions Manual** for the chapter you are interested in. Please also note that the PDF of the 3rd edition of the book that we have available for the course may have slightly different exercise numbering than the book website. In this exercise sheet we always refer to exercise numbers from the **newer book**.

The deadline for assignment 1 is **March 31**. The exercises marked with **(*)** must be passed in order to pass the assignment.

Exercises

1. **(*)** (Exercise 2.9 in the course book.) Consider an on-line course registration system that allows students to add and drop classes at a university. Give the multiplicities of the associations between these class pairs:
   (a) `Student`–`Course`
   (b) `Course`–`Section`
   (c) `Section`–`Instructor`
   (d) `Section`–`Room`
2. Exercise 2.12 in the course book.
3. Exercise 2.13 in the course book.
4. (Exercise 3.1 in the course book) Have a look at the `Calendar` and `GregorianCalendar` classes in the standard library. The `Calendar` class is supposed to be a general class that works for arbitrary calendars, not just the Gregorian calendar. Why does the public interface fall short of that ideal? *You can compare your answer with the solution in the student solution manual.*

5. **(*)** (Exercise 3.8 in the course book) Implement a class `Matrix` that represents a matrix of the form

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,c-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,c-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{r-1,0} & a_{r-1,1} & \cdots & a_{r-1,c-1} \end{pmatrix}$$

where r and c are the number of rows and columns of the matrix. Your class should support the following operations:

- Constructs a matrix with a given number of rows and columns.
- Gets and sets the element at a particular row and column position.
- Adds and multiplies two compatible matrices.

As the internal representation, store the elements in a two-dimensional array

```
private double[][] elements;
```

In the constructor, initialize the array as

```
elements = new double[r][c];
```

Then you can access the element at row `i` and column `j` as `elements[i][j]`.

6. **(*)** (Exercise 3.13 in the course book) List three immutable classes from the standard library.

7. **(*)** A *stack* is a simple abstract data type which can be used to collect a set of elements. The two principal operations for a stack are *Push* and *Pop*. The *Push*(e) operation inserts the element e on top of the set of existing elements in the stack and the *Pop* operation removes and returns the element on top of the stack.

   - Implement the stack data type with integer elements and the above two operations. Then implement two extra operations for this data type which work as follows. The *Push*($n, [e_1, \ldots, e_n]$) operation inserts the set of n elements on top of the stack (e_n is placed on top of the stack), and *Pop*(n) removes and returns the n elements from the top of the stack (the implemented method returns the n elements as a list, where considering the order, the element on top of the stack is the last element in the list).
   - Check the correctness of the implemented operations by writing unit tests using **JUnit**.

8. **(*)** (Exercises 3.20 and 3.21 in the course book) Improve the circular array implementation of the bounded queue by growing the `elements` array when the queue is full. Add assertions to check all preconditions of the methods of the bounded queue implementation.

9. **(*)** Re-implement the stack class from assignment 7 this time with message elements (same as elements in message queue) and implement a method `size()` which returns the number of the messages in the stack.

10. Add two following methods to the message queue class (which is a circular array). The first method, `multAdd(m1, m2, ..., mn)`, is an extension of the `add` method which adds a set of elements to the circular array instead of one element. The second method, `multRemove(n)`, is an extension of the remove method which removes `n` elements from the queue instead of one. Check the correctness of your implementation using **JUnit**.

11. (Exercise 4.2 in the course book) In Java, a method call on an object, such as `x.f()` is resolved when the program executes, not when it is compiled, in order to support polymorphism. Name two situations where the Java compiler can determine the exact method to be called before the program executes.

12. **(*)** (Exercise 4.1 in the course book) When sorting a collection of objects that implements the `Comparable` type, the sorting method compares and rearranges the objects. Explain the role of polymorphism in this situation.

13. (Exercise 4.6 in the course book) Write a method

    ```
    public static String maximum(ArrayList<String> a, Comparator<String> c) { ... }
    ```

    that computes the largest string in the array list using the ordering relation that is defined by the given comparator. Supply a test program that uses this method to find the longest string in the list.

14. **(*)** (Exercise 4.9 in the course book) Define an interface type `Filter` as follows:

    ```
    public interface Filter {
       boolean accept(String x);
    }
    ```

    Then supply a method

    ```
    public static String[] filter(String[] a, Filter f) { ... }
    ```

    that returns an array containing all the elements of `a` that are accepted by the filter. Test your method by filtering an array of strings and accepting all strings that contain at most three characters.

15. **(*)** (Exercise 4.14 in the course book)

16. (Exercise 4.22 in the course book)