# Linked Lists, Stacks and Queues

**Table of Contents**



## 1. Session 4 (lab): Stacks and Queues

### 1.1. Implementing queues with linked lists

The main goal of this assignment is to learn how to implement and use queues and stacks with Java. In order to do so, the solution of the following exercises will be based on linked lists.

Write the class `LinkedQueue`. It must implement the interface `Queue`. You can find this interface below.

```
public interface Queue<E> {
    boolean isEmpty();
    int size();
    void enqueue (E info);
    E dequeue();
    E front();
}
```

Use the `Node.java` class internally when writing your solution. Remember that you already worked with this class in the previous lab session. If you do not have your implementation of `Node.java` available, you can use the code below.

```
public class Node<E> {
    private E info;
    private Node<E> next;

    public Node() {
        info = null;
        next = null;
    }

    public Node(E info) {
        setInfo(info);
    }

    public Node(E info, Node<E> next) {
        setInfo(info);
        setNext(next);
    }

    public E getInfo() {
        return info;
    }

    public void setInfo(E info) {
        this.info = info;
    }

    public Node<E> getNext() {
        return next;
    }

    public void setNext(Node<E> next) {
        this.next = next;
```

```
    }
}
```

## 1.2. Robot for Saturn exploration

ESA is developing a project to put an exploration robot on the surface of Saturn. Due to the delay of Earth-Saturn communications, the robot must be able to perform explorations independently. Therefore, besides the remote control from Earth, it is supported by a complex artificial intelligence (IA) system that is executed in the robot itself.

The remote control from Earth programs the next movements the robot must do, and sends them to the robot. The robot saves in a queue the pending movements. Every time the IA module considers that the robot can continue with the exploration, it dequeues the next movement and performs it.

The robot must be able to perform two types of basic movements: displacements (to move forwards and back), and rotations over its vertical axis (right or left). In order to perform a displacement, it is necessary to specify the distance (an integer representing the centimeters, which will be negative in case of moving back). To rotate, the angle (integer representing the grades, which will be a negative amount if the rotation is to the left) must be specified. The movements are represented with the following class:

```
public class Movement {

    public final static short DISPLACEMENT = 1;
    public final static short ROTATION = 2;

    private short type;
    private int magnitude;

    public Movement(short type, int magnitude) {
        this.type = type;
        this.magnitude = magnitude;
    }

    public String toString() {
        if (type == DISPLACEMENT) {
            return "DSP " + magnitude;
        } else if (type == ROTATION) {
            return "ROT " + magnitude;
        } else {
            return "ERR";
        }
    }

    public short getType() {
        return type;
    }

    public int getMagnitude() {
        return magnitude;
    }

    public Movement reverse() {
        return new Movement(type, -magnitude);
    }
}
```

### Section 1

You are part of the ESA engineers team. You are in charge of writing the `MovementProgrammer` class. This class must register in a queue the movements sent from Earth, and provide them to the robot's IA module as it requests them. Code the `MovementProgrammer` class, including its attributes and a constructor with no parameters. Besides, this class must include the following methods:

- `void program(Movement m)`: invoked by the Earth remote control to add a new movement to the movements queue.
- `Movement next()`: returns the next movement the robot has to perform. This next movement will be extracted from the movements queue. The method returns `null` if the queue is empty. The robot's IA module invokes this method every time it is prepared to perform a new movement.
- `int numMovements()`: returns the number of movements stored in the movements queue.
- `void reset()`: deletes all the pending movements, leaving the movements queue in its initial state.

### Section 2

In order to test the movements programmer, you have to write a small interactive program using the command line. The program will be in a continuous loop in which it will read a command in the standard input and perform

it. The commands that can be introduced by the user are:

- `displace` *magnitude*: programs a new displacement movement with the given magnitude (e.g., `displace -30`).
- `rotate` *magnitude*: programs a new rotation movement with the given magnitude (e.g., `rotate -30`).
- `execute`: executes the next movement
- `state`: displays the current number of pending movements.
- `reset`: removes all the current pending movements.
- `exit`: exits the program.

Next you can find a typical session of the use of the program.

```
C:\Users\Alumno>java TestMovementProgrammer
> displace 10
Programmed: DSP 10
> rotate 45
Programmed: ROT 45
> displace -5
Programmed: DSP -5
> rotate -45
Programmed: ROT -45
> displace 40
Programmed: DSP 40
> execute
Executed: DSP 10
> state
Movements remaining: 4
> rotate 20
Programmed: ROT 20
> displace -10
Programmed: DSP -10
> execute
Executed: ROT 45
> execute
Executed: DSP -5
> execute
Executed: ROT -45
> state
Movements remaining: 3
> execute
Executed: DSP 40
> execute
Executed: ROT 20
> execute
Executed: DSP -10
> state
Movements remaining: 0
> execute
ERROR: empty movements queue
> salir
ERROR: syntax error
> exit
```

You can use the following code to help you start writing your program. Make sure of understanding it before starting to modify it.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class TestMovementProgrammer {

    public static void main(String[] args) throws IOException {
        MovementProgrammer programmer = (...)
        boolean exit = false;
        BufferedReader input = new BufferedReader(
                                   new InputStreamReader(System.in));
        while (!exit) {
            System.out.print("> ");
            String command = input.readLine();
            if (command != null) {
                String[] parts = command.trim().split("\\s");
                if (parts.length == 1 && parts[0].equals("exit")) {
                    (...)
                } else if (parts.length == 2 && parts[0].equals("displace")) {
                    (...)
                } else (...)
                (...)
            } else {
                exit = true;
            }
        }
    }
}
```

# 2. Homework

## 2.1. Implementing stacks with linked lists

### Section 1

Code the `LinkedStack` class. It must implement the `Stack` interface, which code is shown below.

```
public interface Stack<E> {
    boolean isEmpty();
    int size();
    void push (E info);
    E pop();
    E top();
}
```

## 2.2. Advanced robot for Saturn exploration

When the IA module of the ESA robot in Saturn detects the robot ended up in a dangerous position, it can undo the last movements in order to return to a previous safe location. In order to do so, ESA needs to enhance the movements programmer so that it allows to undo the movements executed more recently in the inverse order (similarly to the undo functionality of a text editor).

### Section 1

Code a class called `AdvancedMovementProgrammer`, which will extend `MovementProgrammer` class from previous exercise. This new class adds the following new functionalities with respect to the `MovementProgrammer` programmer.

- Besides the movements queue, the new programmer has a stack where it stores every movement performed (the most recent movement will be on top of the stack). Every time the `next()` method returns a movement different from `null`, this movement must be stored in the stack.
- The new programmer provides an additional method, `Movement undo()`, which returns the movement that allows to undo the last movement performed. In order to do so, the method removes the top method from the stack and returns the inverse one. For instance, if the last movement performed was "DSP 10", the method gets this movement from the top of the stack, and returns the inverse one: "DSP -10". Besides, for security reasons, the method empties the pending movements queue, since those movements could make no sense once the last movement was undone. If the stack of performed movements is empty, this method must return `null` and leave the pending movements queue unmodified.
- The new class provides an additional method, `int numUndoableMovements()`, which returns the number of movements that can be currently undone (i.e., the stack size).

### Section 2

Using the code in `TestMovementProgrammer` class as starting point, write the `TestAdvancedMovementProgrammer` class. This new class must test the functionalities of `AdvancedMovementProgrammer` class. The new program will provide the same commands than in the previous case, and also the following ones:

- Add the command `undo`. This new command undoes the last movement performed and shows the corresponding inverse movement.
- Modify the command `state` so that it indicates both the number of pending movements in the queue as well as the number of movements that can be undone.

Next you can see a typical session of the program as an example.

```
C:\Users\Alumno>java TestAdvancedMovementProgrammer
> rotate 45
Programmed: ROT 45
> displace 10
Programmed: DSP 10
> state
Movements remaining: 2; undoable movements: 0
> execute
Executed: ROT 45
> execute
Executed: DSP 10
> state
Movements remaining: 0; undoable movements: 2
```

```
> undo
Executed: DSP -10
> displace 50
Programmed: DSP 50
> execute
Executed: DSP 50
> state
Movements remaining: 0; undoable movements: 2
> undo
Executed: DSP -50
> undo
Executed: ROT -45
> undo
ERROR: no movement to undo
> exit
```