Learning goals

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

- Create full-Java programs, without any reliance on Groovy. These programs must be completely object-oriented and launch from the main method of one of the classes.
- Understand what the keyword `static` means and how it works.
- Understand how the `do...while` loop structure works and when to use it instead of a `while` loop.
- Be able to create linked lists and travel through their elements.

You should be able to finish most of non-star exercises in the lab. Remember that star exercises are more difficult. **Do not try them unless the other ones are clear to you**.

1 Instance counter

Complete the example given in the notes with a class called `Spy`. Your class must have:

- one and only one `static` variable, and `int` called `spyCount`.
- an instance variable of type `int` for the spy's ID.
- a constructor method that receives the ID of the spy as an argument, increases `spyCount(int)` by one, and prints on the screen the ID of this spy plus the total number of spies so far.
- a `die()` method that prints on the screen "Spy XX has been detected and eliminated" (where XX is the spy's ID), decrements the spy counter and prints on the screen the total number of spies so far.
- a main method in which several objects of class `Spy` are created and some of them killed (their method `die()` is called).

Observe how the static variable is accessed by different objects both to increment and to decrement it.

2 do {practice} while (!understood);

Make a class that implements a method that reads a list of marks between 0 and 100 from the user, one per line, and stops when the user introduces a -1. The program should output at the end (and only at the end) how many marks there were in total, how many were distinctions (70–100), how many were passes (50–69), how many failed (0–49), and how many were invalid (e.g. 150 or -3). **Use `readLine()` exactly once**. The output may look similar to this example:

```
Input a mark: 13
Input a mark: 45
Input a mark: 63
Input a mark: 73
Input a mark: 101
Input a mark: 45
Input a mark: 18
Input a mark: 92
Input a mark: -1
There are 7 students: 2 distinctions, 1 pass, 4 fails (plus 1 invalid).
```

3 Singly-linked lists

Create a linked list following the “hospital and patients” example of the notes:

1. Create the linked list and add several elements to it (around 10 is fine).
2. Go through the list printing out the content of each element.
3. Delete a couple of elements from the list.
4. Print the elements in the list again. Check that the deleted elements have been deleted. Check that you can delete the first element too.

Hint: Note that you cannot delete the first element from inside the list because you need to update the pointer `patientListStart`. To add or delete the first element of a list, you must do it from outside the list.

4 Doubly-linked lists

A doubly-linked list is a dynamic list in which each element is connected to two other elements instead of just one. The one before and the one after it. In this exercise, you must create a doubly-linked list using the “hospital and patients” example as a starting point.

1. Create the doubly-linked list and add several elements to it (around 10 is fine).
2. Traverse it forwards and backwards printing out the content of each element.
3. Delete a couple of elements from the list.
4. Traverse it forwards and backwards printing out the content of each element.
5. Add a new element to the list. Try to delete an element that is NOT in the list.
6. Traverse it forwards and backwards printing out the content of each element.

Hint: First you need to add a new field to Patient for the pointer going “backwards”. Then you need to modify the *add* and *delete* methods to make sure you do not have loose pointers.

5 Circular lists

A circular list is a dynamic list in which there is a beginning but there is no end: the last element is pointing back to the first element. Circular lists can be singly- or doubly-linked. In this exercise, you must create a singly-linked circular list using the “hospital and patients” example as a starting point.

1. Create the circular list and add several elements to it (around 10 is fine).
2. Traverse it forwards for one complete loop. While you go around the list, print out the content of each element. How do you know that you have reached the end of the list when you never find a `null` pointer?
3. Delete a couple of elements from the list.
4. Traverse it again printing out the content of each element.
5. Add a new element to the list. Try to delete an element that is NOT in the list.
6. Traverse it again printing out the content of each element.

6 Queues (*)

A queue is a dynamic structure that implements these methods:

insert(...) inserts an element at the beginning of the queue.

retrieve(...) remove an element from the *end* of the queue.

size() returns the current size of the queue.

Queues are heavily used in computing: communication buffers, incoming request to web servers, read/write requests to hard disks, etc.

Implement a queue of integers. This could represent requests to a hard drive to read from different sectors. Implement the three methods listed above.

Then write another class that creates a queue, and makes several `insert()`, `retrieve()`, and `size()` calls. Check that the values you get are consistent. The output of the program could look like this:

```
There are 0 requests in the queue.
Inserting request 5...
Inserting request 8...
Inserting request 12...
There are 3 requests in the queue.
Retrieving request 5... done.
Inserting request 13...
There are 3 requests in the queue.
Retrieving request 8... done.
Retrieving request 12... done.
There are 2 requests in the queue.
Retrieving request 13... done.
etc...
```

7 Stacks (*)

A stack is a dynamic structure that implements these methods:

push(...) inserts an element at the beginning of the stack.

pop(...) remove an element from the *beginning* of the stack.

empty(...) returns true if there are no elements on the stack, false otherwise.

Stacks are heavily used in computing. The method call stack that stores the variables for each method in a program is just one well-known example.

Implement a stack of integers. This could represent requests to a hard drive to read from different sectors. Implement the three methods listed above.

Then write another class that creates a stack, and makes several `push()`, and `pop()` calls. Make sure you check the size of the stack before popping elements out. Check that the values you get are consistent. The output of the program could look like this:

```
Pushing 5...
Pushing 8...
Pushing 12...
Popping... it’s a 12
Pushing 13...
Popping... it’s a 13
Popping... it’s a 8
Popping... it’s a 5
Stack is empty
```

8 From array to list

Create a static method that takes an array of integers and returns a linked list of integers in the same order as the original array. You can put this method in a class called `ListUtilities` so that you can easily use it for the following exercises.

Initialising an array with 15-20 elements is very easy with curly-brackets-notation, so it is common to create an array and then convert it into a dynamic structure automatically.

9 A sorted list

Create a linked list of integers that is automatically sorted.

Everytime you add an element, make sure you introduce it at the right place in the queue so that the queue is always sorted from lower to higher numbers.

Create the list, insert several numbers into it (around 15 is fine) and then go through the list printing the value of each element. Check that the elements are correctly sorted.

10 Bubble sort (*)

In this exercise, you will implement the *bubble sort* algorithm and use it to sort an unsorted linked list of integers.

The bubble sort algorithm is the simplest sorting algorithm there is:

1. Take the first element. Compare it with the second element. If it is greater, make them change positions (first becomes second and viceversa); otherwise, do nothing.
2. Move to the next element. Repeat the process (i.e. compare 2 and 3, then 3 and 4, up to the last element).
3. Now you have the highest element at the end of the list.
4. Move back to the beginning and repeat the whole process until your list is sorted.
5. Once you pass through the list without swapping elements, your list is sorted.

Put your implementation in a static method in a class, maybe `ListUtilities.bubbleSort(List)`. The name of the algorithm comes from metaphorical bubbles always moving up in a liquid.

Try your method with different lists and see how long it takes to sort them. You can use the static method `System.currentTimeMillis()` to print on the screen the current time (measured in milliseconds since 1st January 1970).

11 Cocktail sort (*)

Cocktail sort is a combination of two bubble sorts. First the list is traversed in one direction to move the highest element to the end, and then it is traversed in the opposite direction to move the lowest element to the beginning. This bidirectional process is repeated until there are no swaps in the list, which shows that the list is sorted.

Implement cocktail sort in a static method in a class, maybe `ListUtilities.cocktailSort(List)`. The name of the algorithm comes from a metaphorical cocktail shaker moving up and down, up and down...

Try your method with the same lists as bubble sort and see how long it takes to sort them.

12 QuickSort (**)

This is a popular sorting algorithm that is very fast in most situations. Once you have finished all the other exercises, look up in wikipedia how this method works and make your own implementation (put it in `ListUtilities.quickSort(List)`). Then compare its performance with the other two methods. In order to see significant differences, you will need to use lists of some length.