Learning goals

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

- Understand the importance of automated testing.
- Write your own tests for classes already defined.
- Execute your own tests for classes already defined.

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

1 Install JUnit 4

Use your favourite search engine to find JUnit ("JUnit download" will do, and probably just "JUnit" will do too).

- Download the latest stable version of JUnit 4.
- Unzip it in your drive. You will find a JAR file, with a name similar to `junit-4.10.1.jar`.
- Place the JAR file in a place where you can find it. It is probably a good idea to create a folder "lib" where you place all the external libraries that you use (e.g. `h:\lib` on windows or `/opt/lib` on Unix systems).
- Now you can add the JAR file to your classpath at the command line (as in the notes) or by modifying the environment variable CLASSPATH.

2 Testing mathematical functions

On Day 7 you implemented a simple hash. Write a battery of tests to verify its behaviour, paying special attention to border cases.

Hint: Implement a loop that tries a fair amount of random numbers (around two thousand, for the purposes of this exercise) and verify that the output is within the range.

3 Practice "Find bugs once"

a)

The method `getInitials(String)` has a bug! If you introduce a name with more than one space between words, it throws an exception.

Create a class that contains the method `getInitials(String)` as described in the notes. Create also the test class as described in the notes.

Then follow the "find bugs once" algorithm: reproduce the bug manually, reproduce the bug programmatically by adding a new test to the testing class, then fix the bug and check that all tests pass.

b)

Another programmer has created a simple program for personal accounting. This program asks the user for the number of bills that have not been paid yet and calculates the total debt. Look at the Accounting and Bill classes below.

The programmer comes to see you because the program does not work as expected: it says that there is one bill too many, and the calculated total debt is wrong too. The programmer asks for your help. You decide to use the "find bugs once" algorithm to find the bug, create a test that reproduces the bug, and then fix the bug once and for all.

This is a common situation in companies. There is some software that has been developed in the past (maybe by programmers that have left the company since) and it needs fixing. The programmers responsible of fixing the program have never seen the code before, they are just told what it should do.

```
/**
 * A program for simple personal accounting.
 *
 * Asks the users about their bills, and then
 * says how many bills there are, and what is the
 * total debt.
 */
public class Accounting {
    /**
     * The first element of the list of bills
     */
    private Bill firstBill = null;

    public static void main(String[] args) {
        Accounting acc = new Accounting();
        acc.launch(args);
    }

    private void launch(String[] args) {
        String concept = "";
        int amount = 0;
        do {
            System.out.println("What's your next bill (type \"END\" to finish)?");
            System.out.print("  Concept: ");
            concept = System.console().readLine();
            if (!concept.equals("END")) {
                System.out.print("  Amount: ");
                String strAmount = System.console().readLine();
                amount = Integer.parseInt(strAmount);
            }
            Bill newBill = new Bill(concept, amount);
            addBillToList(newBill);
        } while (!concept.equals("END"));
        int count = 0;
        int totalDebt = 0;
        for (Bill current = firstBill; current != null; current = current.getNextBill()) {
            count++;
            totalDebt += current.getAmount();
        }
        System.out.println("You have " + count + " bills unpaid (total debt: " + totalDebt + ")");
    }
```

Figure 1: Class Accounting (1 of 2)

```
    private void addBillToList(Bill bill) {
        if (firstBill == null) {
            firstBill = bill;
            return;
        }
        Bill current = firstBill;
        while (current != null) {
            if (current.getNextBill() == null) {
                current.setNextBill(bill);
                return;
            }
            current = current.getNextBill();
        }
        return;
    }
}
```

Figure 2: Class Accounting (2 of 2)

```
public class Bill {
    private String concept;
    private int amount;
    private Bill next;

    public Bill(String concept, int amount) {
        this.concept = concept;
        this.amount  = amount;
        this.next    = null;
    }
    public String getConcept() {
        return concept;
    }
    public int getAmount() {
        return amount;
    }
    public void setNextBill(Bill bill) {
        next = bill;
    }
    public Bill getNextBill() {
        return next;
    }
}
```

Figure 3: Class Bill

Usually the first step is to *refactor* the code to make it easier to test; this may involve breaking up methods and classes into smaller, simpler methods and classes. For example, you may want to separate the "getting user input" and the "create new bill" functionalities in method `launch()` above, so that you can test the latter with JUnit; you can never test user input in an automated way, but you can test what you do with that input by creating fake user data and passing it to your methods.

4 Test implementations of a given interface

You already know that an interface is a way of describing the behaviour of a class without any knowledge about the implementation details. Sometimes, one party provides the interface of a component and the other party implements the interface. This is very common in big projects, where small teams of programmers make parts of a bigger program (e.g. web browsers, word processors, multiplayer games), and the different modules need to communicate with each other. Defining clear and simple interfaces is usually the first step in the design, as it allows different teams to work in parallel and then bring their code together.

Sometimes, the first party does not only define the interface, it also implements the tests that the implementation (i.e. the class that implements the interface) must pass[1]. This is a good idea when the development is sub-hired to an external company.

Take the role of a project leader and implement the tests for two of the interfaces you have implemented in past weeks.

4.1 Stack

The notes from Day 7 implemented a Stack interface in two different ways. Create a battery of tests that verify that the classes implementing the interface is working as expected. Use it to test both implementations.

4.2 Queue

You implemented a Queue interface —maybe in two different ways— on Day 7. Create a battery of tests that verify that the class(es) implementing the interface work/s as expected.

4.3 Set (*)

You implemented a Set interface —possibly in two different ways— on Day 8. Create a battery of tests that verify that the class(es) implementing the interface work as expected.

5 Testing dynamic structures

Write batteries of tests to verify the functionality of the dynamic structures you have created in past weeks:

- doubly-linked list (day 6)
- circular list (day 6)
- simple map (day 7)
- sorted list (day 6)

Make sure that you test border cases, including situations like:

- Adding the first element.
- Removing the last element.
- Adding the first element and then removing it... and then adding another one.

You should have time to do at least one the four cases in the lab.

[1] When this approach is taken to its logical conclusion, we are talking about Test-Driven Development as we will see very soon.

6 More tests (**)

If you have finished with the other exercises, write additional batteries of tests for other programs (in particular, the exercises marked with a star) that you have written in past weeks. Some exercises that provide a harder challenge to test properly are:

- the anti-aircraft game from day 5
- any of the sort algorithms from day 6
- any of the unfair queues from day 7
- the hash-table (day 7)
- deletion of elements in a tree (day 8)
- re-balancing of a tree (day 8)
- the abstract syntax tree (day 8)
- the pseudo-git tree (day 8) (**)