Software and Programming II (SP2) — Lab sheet 10

JUnit and Dealing with Exceptions

2018/19

1. Consider the following class (also on Moodle), which provides a simple implementation to compute the n-th Fibonacci number (`https://en.wikipedia.org/wiki/Fibonacci_number`).

```
package fibonacci;

/**
 * Class with a static method to compute the n-th Fibonacci number fib(n)
 * for n >= 0 according to the definition
 *
 * fib(0) = 0
 * fib(1) = 1
 * fib(n) = fib(n-1) + fib(n-2) for n >= 2
 */
public class Fibonacci {

     /**
      * Straightforward recursive implementation for the n-th Fibonacci number.
      *
      * @param n expected to be non-negative
      * @return the value of the n-th Fibonacci number fib(n)
      * @throws IllegalArgumentException if n < 0
      */
     public static long fib(int n) {
         if (n < 0) {
             throw new IllegalArgumentException("Illegal negative value " + n);
         }
         if (n == 0) {
             return 0;
         }
         if (n == 1) {
             return 1;
         }
         return fib(n-1) + fib(n-2);
     }

     public static void main(String[] args) {
         final int N = 46; // makes the method take a while on my machine
         long fibN = fib(N);
         System.out.println("fib(" + N + ") = " + fibN);
     }
}
```

Write a *JUnit test suite* for JUnit 4 with suitable unit tests for the method

`public static long fib(int n)`

Try to achieve *test coverage* for the method, i.e., try to come up with a set of test cases that together cover all (reachable) code in the method. (This also includes testing that the `IllegalArgumentException` is triggered.)

In particular, introduce two test cases for the input 46 (expected result: 1836311903), one with a *timeout* of 600 seconds and one with a timeout of 1 second (if the second test case passes, you are using one fast machine!).

2. Consider the following code.

```
 1 package debugging;
 2 public class SomeExample {
 3
 4     /**
 5      * Returns the sum of the lengths of the non-null entries in strings.
 6      *
 7      * @param strings must not be null, but may contain null
 8      * @return the sum of the lengths of the non-null entries in strings
 9      * @throws NullPointerException if strings is the null reference
10      */
11     public static int sumLengths(String[] strings) {
12         int result = 0;
13         for (String s : strings) {
14             result += s.length();
15         }
16         return result;
17     }
18
19     /**
20      * Computes and prints the sum of the lengths of the non-null entries in
21      * words.
22      *
23      * @param words must not be null, but may contain null
24      * @throws NullPointerException if words is the null reference
25      */
26     public static void process(String[] words) {
27         int sum = sumLengths(words);
28         System.out.println("The sum of the lengths is " + sum);
29     }
30
31     public static void main(String[] args) {
32         String[] myWords = new String[3];
33         myWords[0] = "Hello";
34         myWords[1] = null; // myWords /contains/ the null reference
35         myWords[2] = "World";
36         process(myWords);
37         System.out.println("Take 2.");
38         String[] noWords = null; // noWords /is/ the null reference
39         process(noWords);
40         System.out.println("Bye!");
41     }
42 }
```

The code compiles without problems. However, there are two "bugs" (programming errors) in the code, which make it behave incorrectly according to the specification given in the methods' documentation comments. Your task is to find the places where the program works incorrectly and to repair the program.

Hint:

When we run the code, we get the following output on the screen (called a "stack trace"):

```
1 Exception in thread "main" java.lang.NullPointerException
2         at debugging.SomeExample.sumLengths(SomeExample.java:14)
3         at debugging.SomeExample.process(SomeExample.java:27)
4         at debugging.SomeExample.main(SomeExample.java:36)
```

This stack trace tells us several things.

1. Line 1 of the stack trace tells us that the program "crashed" with a `NullPointerException`. So either a `NullPointerException` was thrown explicitly (this is usually not the case), or an attribute or method was accessed via the `null` reference.
2. Line 2 of the stack trace tells us that the program was running the method `sumLengths` and that it was running (the Java bytecode corresponding to) the instruction(s) in line 14 of the file `SomeExample.java`. So the illegal access to an attribute or method of the `null` reference must have been somewhere in line 14.
3. Line 3 of the stack trace tells us that the method `sumLengths` had been called in line 27 of `SomeExample.java` in the method `process`.
4. Line 4 of the stack trace tells us that the method `process` had been called in line 36 of `SomeExample.java` in the method `main`.
5. There is no further output, so this means that the program had been started from the method `main` (this is usually the case, but not always).

Such a stack trace can span dozens of lines. However, it is often enough to look at the first three or four lines of a stack trace to get an idea of what went wrong.