Programming in Java (24/25)

– Exercises Day 10 –

Learning goals

Before the next day, you should achieve the following learning goals:

- Understand what exceptions are and how they are used.
- Use stack traces from exceptions to find bugs in your program.
- Create your own exceptions, both checked and runtime.
- Catch exceptions by means of `try`/`catch`.
- Read from text files.
- Write to text files.

1 Exception

Look at the following method and check whether you think there is anything wrong with it from the Java compiler's perspective. Then check your answer with the actual Java compiler.

```
1  public static <E> void addElement(java.util.List<E> list, E newElement) {
2      try {
3          list.add(newElement);
4      } catch (Exception e) {
5          e.printStackTrace();
6      } catch (NullPointerException e) {
7          e.printStackTrace();
8      }
9  }
```

2 Detective work

Consider the following code.

```java
package pij.day10;
public class SomeExample {

    /**
     * Returns the sum of the lengths of the non-null entries in strings.
     *
     * @param strings must not be null, but may contain null
     * @return the sum of the lengths of the non-null entries in strings
     * @throws NullPointerException if strings is the null reference
     */
    public static int sumLengths(String[] strings) {
        int result = 0;
        for (String s : strings) {
            result += s.length();
        }
        return result;
    }

    /**
     * Computes and prints the sum of the lengths of the non-null entries in
     * words.
     *
     * @param words must not be null, but may contain null
     * @throws NullPointerException if words is the null reference
     */
    public static void process(String[] words) {
        int sum = sumLengths(words);
        System.out.println("The sum of the lengths is " + sum);
    }

    public static void main(String[] args) {
        String[] myWords = new String[3];
        myWords[0] = "Hello";
        myWords[1] = null; // myWords /contains/ the null reference
        myWords[2] = "World";
        process(myWords);
        System.out.println("Take 2.");
        String[] noWords = null; // noWords /is/ the null reference
        process(noWords);
        System.out.println("Bye!");
    }
}
```

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 pij.day10.SomeExample.sumLengths(SomeExample.java:14)
3         at pij.day10.SomeExample.process(SomeExample.java:27)
4         at pij.day10.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 a field 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.

This kind of detective work is an important part of software development. Many bugs will become visible when the program crashes with some runtime exception and a stack trace on the screen. It is then the programmer's task to find out where the program needs to be changed so that it works (more) correctly. Reasons can be:

1. The exception happens in one of your own methods (i.e., line 2 of the stack trace, right below the name of the exception, shows one of your own methods). This means that your code *directly* leads to an exception. For example: if you write

```
1         Object o = new Object();
2         String s = (String) o; // contrived example
```

   and run the code, line 2 will throw a `ClassCastException`. In this case, you need to repair the code that you have written.

2. The exception happens in existing code (that has usually been written by others). This can have two reasons:

   (a) Your code *is using existing classes or methods in a way they should not be used according to their Java Doc*.

   For example:

```
1        String s = "abc";
2        String shorter = s.substring(-1);
```

   will throw an `IndexOutOfBoundsException`.

   The Java Doc of method `substring(int)` on `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html#substring(int)` says:

   "Throws:
   IndexOutOfBoundsException – if beginIndex is negative or larger than the length of this String object."

   Since we used a negative value for the actual parameter of method `substring(int)`, it was us who caused the `IndexOutOfBoundsException` by using the method with an unsuitable actual parameter.

   (b) Your code *is using the existing code as required by its Java Doc, but there is a bug in the existing code so that it does not do the job as described by its Java Doc*.

   For example, if

```
1        String s = "abc";
2        String shorter = s.substring(1);
```

   were to throw an `IndexOutOfBoundsException`, a careful look at the documentation on `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html#substring(int)` would show that this should not have happened (the result of the method call should have been `"bc"`) and that therefore there would be a bug in method `substring(int)`.

   In this example, the code indeed puts a reference to `"bc"` into the variable `shorter` as required, and it does not throw an `IndexOutOfBoundsException`. In general, it is unlikely (but possible!) that the Java libraries will behave differently from what their Java Doc claims. So if an exception happens in Java library code, we should first check whether we are using the library code correctly before we investigate the details of the library code implementation.

   However, not all existing code has gone through as much testing and as many quality checks as the Java libraries, so it is very much possible that we may find bugs in existing code in this way.

3 Code flow

Look at the following code and write the code flow (use the line numbers to indicate which lines are executed in which order) in the following situations:

- `userInput` is 0.
- `userInput` is 2.
- `userInput` is 4.
- `userInput` is 6.

```
1   public void launch(int userInput) {
2       List<Integer> intList = new ArrayList<Integer>();
3       intList.add(1);
4       intList.add(2);
5       intList.add(3);
6       intList.add(4);
7       intList.add(5);
8       intList.add(6);
9       try {
10          intList.remove(0);
11          System.out.println(intList.get(userInput));
12          intList.remove(0);
13          System.out.println(intList.get(userInput));
14          intList.remove(0);
15          System.out.println(intList.get(userInput));
16          intList.remove(0);
17          System.out.println(intList.get(userInput));
18          intList.remove(0);
19          System.out.println(intList.get(userInput));
20          intList.remove(0);
21          System.out.println(intList.get(userInput));
22          intList.remove(0);
23          System.out.println(intList.get(userInput));
24      } catch (IndexOutOfBoundsException e) {
25          e.printStackTrace();
26      }
27  }
```

Here `List<E>` and `ArrayList<E>` are from package `java.util`. Incorporate this code into a simple class to verify your answers.

4 Error handling on user input

a)

Write a program that reads 10 numbers from the user and then prints the mean average. If the user inputs something that is not a number, the program should complain and ask for a number again until 10 numbers have been introduced.

b)

Modify the program so that it first asks how many numbers the user wants to enter, and then asks for the numbers. The computer should complain if the user enters something that is not a number in both cases. Use methods to prevent code repetitions.

5 Prime divisors

Create a class `PrimeDivisorList`. Integers (as in class `Integer`) can be added to / removed from the list. If a **`null`** number is passed to the `add(Integer)` method, a `NullPointerException` must be thrown. If a non-prime number is added, an `IllegalArgumentException` must be thrown.

Override the method `toString()` so that it returns something like:

```
[ 2 * 3^2 * 7 = 126 ]
```

for a list containing one 2, two 3, and one 7.

You can base your class on classes and interfaces from the Java Collections Library.

6 Your first exceptions

Create two exceptions, one checked exception and one runtime exception. Then create a simple class that will throw each of them in two different situations, according to user input:

1. inside a **`try`** block.
2. outside a **`try`** block.

Note: Two exceptions times two situations means four different inputs from users. Create the new exceptions with four different messages (using the appropriate constructor), e.g., "I am a checked exception and I have been thrown out of a try block".

Assuming you do all of the above inside the `launch()` method of your class, did you have to make any changes to the method's declaration?

7 ls

The names of this exercise and of the following ones are inspired by Unix tools (available, e.g., on Linux and macOS) of the same name that have similar functionality.

Write a program that shows the names of the files in the current directory on screen. (Hint: look at methods from class File).

8 mkdir

Write a program that takes a name from the user at the command line and creates a directory with that name. Remember that the only argument of method `main` is an array of Strings, each of them an argument written after the name of the class. For example, if you write

```
java MyClass this That 0
```

on the command line, the elements in `args` will be three strings `"this"`, `"That"`, and `"0"`.

9 cat

a)

Write a program that takes a name from the user at the command line. If a file with that name exists, the program must show its contents on screen. If it does not, the program must say so.

b) (*)

Modify the program so that it takes many file names at the command line, and then shows them all one after the other.

10 cp

a)

Write a program that takes two names from the user at the command line. If a file with the first name exists, the program must copy it line by line into a file with the second name.

If the first file does not exist, the program must say so. If the second file does exist, the program must ask the user whether to overwrite it or not, and act accordingly.

b) (*)

Modify the program so that it takes many file names at the command line. When this happens, the last name must be a directory (otherwise, your program should complain). If it is a directory, your program has to copy all files (i.e., the other arguments) into that directory.

11 tr (*)

Write a program that takes a name and two strings from the user at the command line. If a file with that name exists, the program must show its contents on screen, but substituting any occurrence of the first string by the second string (as if search-and-replace had been done). If the file does not exist, the program must say so.

12 sort (*)

Write a program that takes a name from the user at the command line. If a file with that name exists, the program must show its contents on screen, but with the lines shown in lexicographic order. If the file does not exist, the program must say so.

Hint: Remember that Strings in Java implement the interface `Comparable<String>`.

13 uniq (*)

Write a program that takes a name from the user at the command line. If a file with that name exists, the program must show its contents on screen, but removing duplicates lines (showing on screen only one line for each set of duplicated lines). If the file does not exist, the program must say so.

14 find (*)

Write a program that takes a name from the user at the command line and returns all files with the same name in the current folder *and in any subdirectory*.

15 Temperature averages

Write a program that reads a file from disk in *comma-separated values* format (CSV). Here this means that every line contains a list of numbers separated by commas.[1] The program must output the average for every line plus the average for the whole file. A file may look like the following:

```
25,24,20,18,15,13,14,13,15,17,19,21
25,25,24,20,18,15,13,14,13,15,17,19
21,25,25,24,20,18,15,13,14,17,19,21
25,25,24,20,18,15,13,14,13,15,17,19
21,25,25,24,20,18,15,13,14,13,15,17
21,25,25,24,20,18,15,13,14,13,15,17
19,21,25,25,24,20,18,15,13,14,13,15
17,19,21,25,25,24,20,18,15,13,14,13
```

[1] In general, the CSV format allows also for other data types, and it is often used for exchange of tabular data between spreadsheet applications. If you are interested, here is more information on CSV: `https://en.wikipedia.org/wiki/Comma-separated_values`

16 Binary cp (**)

Write a program that takes two names from the user at the command line. If a file with the first name exists, the program must copy it into a file with the second name. If the first file does not exist, the program must say so. If the second file does exist, the program must ask the user whether to overwrite it or not, and act accordingly.

This is the same exercise as above with an important difference: it must be able to copy binary files (use `InputStream` instead of `Reader`, etc). Try it with `.class` or `.exe` files and check that the copies work exactly like the originals.