Assignment 01: Maze Explorer COSC 102

Learning Outcomes

This homework is an application of *procedural programming*, the programming paradigm you learned in COSC101 and is being reviewed in this first part of COSC102. This first implementation puts in practice your readings of the textbook (BJP) and the design concepts we are emphasizing during lectures and labs. Read this entire document before the start of lab tomorrow.

In this first Java assignment you will practice

1. using objects (such as `String` and `Scanner`),
2. creating and traversing Java most basic data structure, which are arrays,
3. dealing with input—from files and from standard input, i.e., the console—and generating output, i.e. text on the console; input/output also take the form of function parameters and return values, and
4. writing helper functions, which are designed according to the SOFA properties:
   - **S**hort—they execute a short function body,
   - **O**ne—they do only one thing,
   - **F**ew—they take few parameters (two or three is good), and
   - **A**bstraction—they are based on an abstraction.

Specifically, in Java you will implement code

- to represent a maze based on an input file,
- to generate its ASCII “art” output on the console and
- to explore the maze’s content according to a path entered an user at the console.

Tomorrow in lab, you will have the opportunity to ask questions to your lab instructor after being assigned your homework partner. Annotate this document, writing ideas you have on how to solve a task; do **not** write any code on the computer. You are required to solve Part I using the pair programming development method, an efficient and collaborative practice commonly used in the industry. You will demo the functionalities for Part I during next week lab period. The following week you will solve independently Part II.

1 Representing Mazes

1.1 Maze Construction

Description. Your program takes as input the information that define a maze so as to display it on the console according to the provided format described below.

The maze is defined by

- a size and
- two types of elements found within the maze boundaries:

1. treasures and
2. rectangular obstacles.

Ultimately[1] the maze information will be provided by a file input. For example, we give you two input files `plan1.txt` and `plan2.txt` to help you write your program incrementally and start testing its correctness. Open them in a plain text editor, such as `jEdit` or `atom`, to examine their content. The content of `plan1.txt` is as follow

```
3 5
$ 1 2
* 3 4 3 4
```

while `plan2.txt` is

```
8 12
$ 2 9
* 1 10 4 10
* 2 2 3 4
* 5 5 5 5
$ 9 1
* 7 7 8 12
```

Each input maze file respects the following three rules.

- The first rule is that the top line of the file contains two integers, representing the number of rows and columns (respectively) for the boundary of the maze.

  From the first line of `plan1.txt`

  ```
  3 5
  ```

  the maze is made of three rows and five columns, such as it is displayed in ASCII as follow:

  ```
  +-----+
  |.....|
  |.....|
  |.....|
  +-----+
  ```

- Each subsequent line contains a character followed either by two integers or by four integers.

  1. In the two integers case, the first character, $, identifies a treasure found at the location specified by the two integers. The first number indicates the row and the second the column of the maze cell where the treasure is to be found.
  2. In the four integers case, the first character, *, identifies obstacles covering a rectangular region within the maze. The first two numbers specify the row and column (respectively) of the upper-left corner of the obstacle; the last two numbers specify the row and column of the bottom-right corner.

     Row and column numbers are assumed to start at 1. You may assume that the input files are such that everything is fully contained in the maze boundaries and that no maze elements overlap.

  From the subsequent two lines of `plan1.txt` the maze contains one treasure and a rectangular obstacle,

  ```
  $ 1 2
  * 3 4 3 4
  ```

[1] We say *ultimately* because before dealing with file input you might design nice helper functions using dummies (aka default hard-coded values as parameters). *Ask if you are confused*: asking is always a good idea.

the ASCII output displayed at the console should be

```
+-----+
|.$...|
|.....|
|...*.|
+-----+
```

Understanding Check. Consider the second provided example, `plan2.txt` content and draw below each of the six elements according to lines subsequent to the top line (lines 2 to 7 are provided on the previous page)

```
+------------+
|............|
|............|
|............|
|............|
|............|
|............|
|............|
|............|
+------------+
```

Example. Executing your program and entering `plan2.txt` as the file input should exactly produced the output below. Make sure to respect the syntax shown as our testing scripts, which check for the correctness of your implementation expect this exact syntax.

```
Please enter the maze file name: plan2.txt
+------------+
|.........*..|
|.***....$*..|
|.***.....*..|
|.........*..|
|....*.......|
|............|
|......******|
|$.....******|
+------------+
```

Programming Task. Define global variables (possibly constants) and write helper methods to store and display the maze data. First tackle its boundaries, then the treasure elements. Finally, take advantage of code reuse to handle the rectangular obstacles.

1.2 Cell Queries

Description. Once a maze has been defined, your program should repeatedly query the user for a cell coordinate input: two integers, row first and column second. According to the cell entered, your program provides feedback: the cell is either outside the maze, a treasure, within an obstacle, or empty. Entering `0 0` terminates the repetition.

Before completing this second task, take a breath and review your code and critic its design. Everything will be easier that way. Think hard about the high-level design of your implementation so far and carefully about the its details. Any repeated lines of code should be examined, can they be abstracted to a function, so to avoid redundancy. Are each function designed according to the SOFA properties? Is there two functions that look the same, i.e., similar parameters and bodies, or serve almost the same purpose? Is there one function that is a more complex version of another one? Could one make use of the other instead? Once you have improve your code, it is easier to add this next feature.

Example. Consider the specific input/output below.

```
Please enter the grid file name: plan2.txt

+------------+
|.........*..|
|.***....$*..|
|.***.....*..|
|.........*..|
|....*.......|
|............|
|......******|
|$.....******|
+------------+

Enter a cell (row and colum; zeros to exit): 10 8
(10, 8) is out of the maze
Enter a cell (row and colum; zeros to exit): 2 9
(2, 9) is a treasure
Enter a cell (row and colum; zeros to exit): 3 3
(3, 3) is within an obstacle
Enter a cell (row and colum; zeros to exit): 4 3
(4, 3) is an empty cell
Enter a cell (row and colum; zeros to exit): 0 0
Goodbye
```

1.3 Checkpoints

Refactoring throughout code development helps writing better program. By the end of Part I, you should have spent enough time revising your code (aka refactoring). The purpose of the collaborative pair programming development strategy is to help you write well-designed code, that is easy to easier to understand and to extend. You want to be proud of your pair achievements!
To help you assess your work take into consideration the following hints.

- Three lines in a function is fine and really means the function does ONE thing. More than eight lines of actual code (omitting `print` calls) might be a sign that the function is doing more than one thing.
- The `main` method should be short.
- An handful of class variables and an handful of constants are expected.
- Ten helper functions or more should not surprise you.