Assignment 2 – PART 1 SU18 CS1027

Part 2 will be released as a second document

Due: Monday July 9th at 11:55 pm via OWL

In this assignment, you will make a java version of the last assignment from CS1026 SU18.
You will create two classes which together allow a program to store, search, remove and filter country data.

Labyrinths (Mazes) can be "solved" by computers using a variety of algorithms. A simple approach is to start at the beginning and search through all open spaces one at a time until the exit or end is found. You will create a program that searches for an exit in a Labyrinth made up of hexagonal tiles, using a stack to keep track of tiles yet to be checked.

Total weight of Assignment 2: 10%

Portion of weight for Part 1: 8% Part 2: 2%

This assignment is to be done individually. You may talk to other students on a high level, but all work must be done independently. All assignments will be run through anti-plagiarism software. First offense penalty will be recorded as an act of academic dishonesty, and a mark of -100% will be awarded. You can also help students work on bugs in their code, but the solution is never to "just copy what I did, here look."

Learning Outcomes:

By completing this assignment, you will gain skills related to

- Stacks and Stack Algorithms
- Writing your own classes in java
- Testing code
- Using Exceptions

Classes from Lectures you will need

- `ArrayStack.java`
- `EmptyCollectionException.java`
- `StackADT.java`

Provided Classes for this Assignment

- `Labyrinth.java` - A class representing a Labyrinth made up of Hexagon tiles. Opens in a graphical window.
- `Hexagon.java` - A class representing the Hexagon tiles in a Labyrinth window.
- `HexComponent.java` - A superclass of Hexagon representing the graphical element
- `HexLayout.java` - A class allowing the Labyrinth to lay out the Hexagon tiles correctly

The main classes you will be working with are Labyrinth and Hexagon. The Labyrinth class is initialized with a file representing the layout of the Labyrinth. Some sample files will be provided.

The API for the provided classes will orient you to their methods: API Reference

Each Hexagon tile can be one of a Wall (black), Start (green), End (red) or Unvisited (cyan) when we build the Labyrinth at the beginning, and as we visit tiles during the solving process, they can become Pushed (magenta) or processed (blue). The tiles will display in different colours so you can track your `SearchForExit`'s progress. The end tile will turn gold when it is found (processed).

Task 1. Create two exception classes

When working with labyrinths and hexagons, there are two situations that come up.

1. UnknownLabyrinthCharacterException
    a. As you can see in the Labyrinth file, the constructor may throw this exception while reading the file describing the labyrinth.
2. InvalidNeighbourIndexException
    a. This is to cover the case where an entity requests a neighbour that is not 0-5 inclusive.

Your task is to create these two exceptions as .java files. Take a look at the EmptyCollectionException to get an idea of what this would look like.

Task 2. SearchForExit.java

You will create a class `SearchForExit` that has a main method only, which implements the Labyrinth solving algorithm here

Algorithm for ***`SearchForExit`***

- Create a Labyrinth object reference
- Try to open a new Labyrinth from one of the Labyrinth files provided
- (The Labyrinth should now be initialized)
- Create a Hexagon reference and get the start Hexagon tile from the Labyrinth
- (Using an array stack of Hexagons, we will solve the Labyrinth.)
- Create the stack and push the starting tile.
- Create a boolean variable that we will use to keep track of whether we have found the end
- Create a reference for the current Hexagon that we will process
- Create a counter to keep track of how many steps it takes us to solve the Labyrinth
- Now, while our stack is not empty, and while we have not found the end
    - pop the top of the stack to get the current hexagon tile
    - increment the step counter
    - if the current hexagon tile is an end tile, make the found end boolean variable true
    - otherwise, for each of the possible six neighbours of the current tile
        - if the neighbour exists and is unvisited
            - push the neighbouring hexagon tile to the Labyrinth solving stack and set the neighbouring hexagon tile to be a pushed tile type
    - set the current hexagon tile to "processed", now that we are done with it
    - Note: each time through we need to update the Labyrinth window with a call to repaint()
- Once we have searched the Labyrinth using the above algorithm, print out the following using a System.out.println or .format:
    - If the end was found, or if it was not found
    - The number of steps that it took to finish
    - How many tiles were still on the stack
- Save the labyrinth's finished state to a file called "processed_" + input file name.
    - Ex. if laby1.txt were the input file, processed_laby1.txt would be the output name

Implementation notes

Exceptions

- Your code must correctly handle thrown exceptions, including the ones you have written.
- To handle the exceptions you need only to inform the user through a console print statement what specifically has happened.

Command Line Argument

You must read the Labyrinth file from the command line. The command line is what computers used to use to run programs rather than having graphical windows, and all systems still have the functionality available (like through the Terminal application on Macs, or the Command Prompt application on Windows). The user could run the SearchForExit program with the following command from a command line (for example):

```
java SearchForExit labyrinth1.txt
```

If you have ever wondered what the "`String[] args`" thing meant in the main method header, this is what it is for; It allows us to read in the text supplied on the command line separated by spaces.

To read a single argument, we look in `args[0]`.

```
try{
            labyrinthFileName = args[0];
```

Note that the example is in a try block. You will need to catch exceptions for trying to read the argument.

Setting up a Command Line Argument when running your program in Eclipse

To get Eclipse to supply a command line argument to your program when it runs, you will need to modify the "Run Configurations".

- Open the "Project"->"Run Configurations" menu item. Something like the following should pop up.
- Be sure the "Java Application->SearchForExit" is the active selection on the left-hand side.
- Select the "Arguments" tab.
- Enter the filename in the "Program arguments" text box.

Non-functional Specifications

- Your program has to be compilable under Eclipse.
- Use Javadoc comments for each class and method.
- Make comments on the major steps of your algorithm
- Use Java conventions and good Java programming techniques (meaningful variable names, conventions for variable and constant names, etc). Indent your code properly.
- Remember that assignments are to be done individually and must be your own work.

Classes to Submit to OWL

- `SearchForExit.java`
- `UnknownLabyrinthCharacterException.java`
- `InvalidNeighborIndexException.java`