CS 1027 Computer Science Fundamentals II
Assignment 3

Due date: November 18, 11:55 pm

1. Learning Outcomes

To gain experience with

- The solution of problems through the use of stacks
- The design of algorithms in pseudocode and their implementation in Java.
- Handling exceptions

1. Introduction

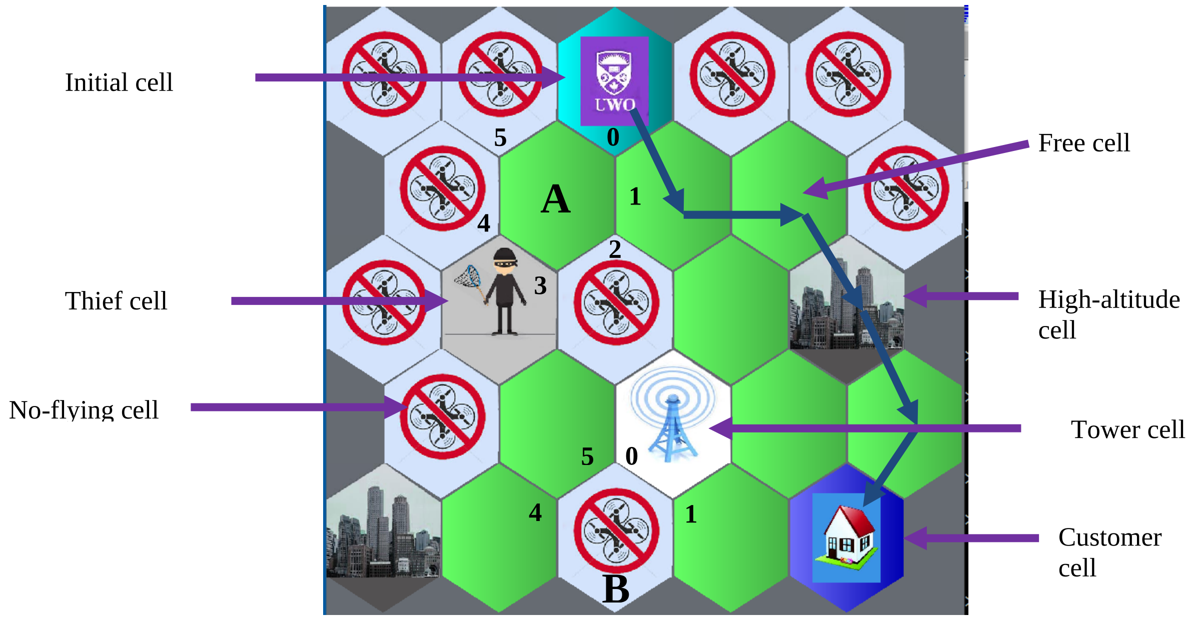
The UWO store has decided to use drones to deliver merchandise to students. For this assignment, you will design and implement a program in Java to compute a path that a drone can follow from the UWO store to a student's house. You are given a map of the city, which for some reason the London cartographer has divided into hexagonal regions or cells. You know the initial map cell, where the UWO store is located, and the destination cell where the student's house is situated. The figure in the following page shows an example of a map divided into cells.

There are 6 types of map cells:

- *Free cells*. The drone can safely fly across these cells.
- *High-altitude cells*. These are map cells where buildings are located. The drone can fly across these cells, but it must increase its flying altitude, which consumes additional power from its battery, reducing its flying time.
- *Thief's cells*. These map cells are locations where thieves lurk. Thieves in these places will try to bring down a drone to steal the merchandise it carries. A drone can go across these cells, but it needs to move very fast, and this also consumes additional power from its battery.
- *Tower cells*. These are map cells where cellular towers with large antennae transmit electromagnetic signals that interfere with the navigation system of the drone. The drone must avoid these cells **and any cells adjacent** to a tower cell.
- *No-flying cells*. The drone is not allowed to fly through these cells, as otherwise the UWO store will be fined.
- *Customer cell*. This is the map cell where the student's house is located. This is the final destination for the drone. The customer cell is a free cell.

Initially the drone is positioned in the free map cell containing the UWO store. As the drone flies toward the customer cell, it might have several choices as to what path to follow; when given the choice between going to a free, a high-altitude, or a thief's cell, the drone will prefer the free one over the other two, and it will prefer a high-altitude cell over a thief's cell.

The following figure shows an example of a map. The starting cell is marked with a purple UWO label. There is one tower cell, a thief's cell and two high-altitude cells. The customer cell is located near the bottom right. A path from the starting cell to the destination is marked. Note that the free cells adjacent to the tower cannot be part of the solution, so the drone has to go over a high-altitude cell.



Each map cell has up to 6 neighboring cells indexed from 0 to 5. Given a cell, the neighboring cell located in its upper right corner has index 0 and the remaining neighboring cells are indexed in clockwise order. For example, in the above figure the neighboring cells of cell A are indexed from 0 to 5 as shown. Note that some cells have fewer than 6 neighbors and the indices of these neighbors might not be consecutive numbers. For example, cell B in the figure has 4 neighbors indexed 0, 1, 4, and 5.

2. Classes to Implement

A description of the classes that you need to implement in this assignment is given below. You **cannot** use any static instance variables. You **cannot** use java's provided *Stack* class or any of the other java classes from the java library that implement collections. The data structure that you must use for this assignment is an array, as described in Section 2.1.

2.1 Class DroneStack.java

This class implements a stack using an array. The header of this class must be this:

public class DroneStack<T> implements StackADT<T>

You can download StackADT.java from the course's website. This class will have the following three private instance variables:

- private T[] arrayStack. This array will store the data items of the stack.
- private int count. This variable stores the number of data items in the stack.
- private int maximumCapacity. The value of this variable indicates the maximum size of the array that stores the items of the stack.

This class needs to provide the following public methods.

- public DroneStack(). Creates an empty stack. The default initial capacity of the array used to store the items of the stack is 10. The value of instance variable maximumCapacity must be set to 1000.

- public DroneStack(int initialCapacity, int maxCap). Creates an empty stack using an array of length equal to the value of the first parameter. The maximum size that the array storing the data items can have is given by the second parameter.
- public void push (T dataItem) throws OverflowException. Adds dataItem to the top of the stack. If the array storing the data items is full, you will increase its length as follows:
    - If the length of the array is smaller than 50, then the length of the array will be increased by a factor of 4.
    - Otherwise, the length of the array will increase by 50. So if, for example, the length of the array is 60 and the array is full, when a new item is added the size of the length of the array increases to 110.
    - If the length of the array is increased to more than the value of maximumCapacity (i.e. maximumCapacity + 1 or larger) an OverflowException exception must be thrown. When creating an object of the class OverflowException, an appropriate String message needs to be passed as parameter to the constructor. We set an upper bound on the capacity of the stack to prevent your program from exhausting the memory of the computer if a bug causes it to enter in an infinite loop.

- public T pop() throws EmptyStackException. Removes and returns the data item at the top of the stack. An EmptyStackException is thrown if the stack is empty. When creating an EmptyStackException a String message must also be passed as parameter.
- public T peek() throwns EmptyStackException. Returns the data item at the top of the stack **without** removing it. An EmptyStackException is thrown if the stack is empty.
- public boolean isEmpty(). Returns true if the stack is empty and it returns false otherwise.
- public int size(). Returns the number of data items in the stack.
- public String toString() throws EmptyStackException. Returns a String containing the data items in the stack listed from the top of the stack to the bottom and separated by a comma. For example if the strings "A", "B", "C" are pushed into the stack in this order, then the method toString must return the string "C,B,A". An EmptyStackException must be thrown if the stack is empty.

You can implement other methods in this class, but they must be declared as private.

2.2 Class Path.java

This class will have an instance variable: DroneMap cityMap.

This variable will reference the object representing the city map where the drone will be flying. This variable must be initialized in the constructor for the class. You must implement these methods:

- public Path (String filename). This is the constructor for the class. It receives as input the name of the file containing the description of the city map, and the initial and destination map cells. In this method you must create an object of the class DroneMap (described in Section 5) passing as parameter the given input file; this will display the map on the screen. Sample input files are also provided on the course's website. Read them if you want to know the format of the input files.
- public static void main (String[] args). This method will first create an object pathObject of the class Path using the above constructor; the parameter for the constructor will be the name of the input file, which is the first command line argument of the program, i.e. args[0] (see Section 4). This method then will invoke method findPath passing as parameter pathObject. Method findPath will return a String, which method main must print. Method main must also catch any exceptions that are thrown when method findPath is invoked and for each exception caught an appropriate message must be printed. Method findPath is not static, how do you invoke it from main?

- public String findPath(Path pathObject). This method will try to find a path from the initial cell to the destination according to the restrictions specified above. The algorithm implemented in this method that looks for a path from the initial cell to the destination **must use a stack of the class DroneStack**. Hints on how to look for this path are given in the next section. The code provided to you will show the path selected by your algorithm as it tries to reach the destination, so you can visually verify how your program works.
  If the algorithm is able to find a path to the destination cell, method findPath must return this String: "Path found of length x cells", where x is the number of cells in the path from the initial cell to the destination (including the initial cell and the destination cell); if no path was found the method must return the String: "No path found".
- public boolean nearTower(MapCell cell). The parameter is the cell where the drone currently is. Class MapCell, described in Section 5, represents the map's cells. Method nearTower returns true if any of the adjacent cells to the current one is a tower cell, and it will return false otherwise. Read Section 5 to learn how to get the cells that are adjacent to the current one.
- public MapCell nextCell(MapCell cell). The parameter is the cell where the drone currently is. This method returns the best cell for the drone to move to from the current one. If several unmarked cells (details about marked and unmarked cells are given in Sections 3 and 5) are adjacent to the current one, then this method must return one of them in the following order:
  - a free cell, if one exists, or the customer cell. If there are several free cells (or several free cells and the customer cell), then the first cell (the one with smallest index) is returned. Read the description of the class MapCell below to learn how neighboring cells are indexed.
  - a high-altitude cell, if no adjacent free cells exist
  - a thief's cell, if there is no neighboring free or high-altitude cell.

  If there are no unmarked cells adjacent to the current cell this method returns `null`.

Note that for each input map there is **only one correct path** that satisfies the specifications stated above. You can write more methods in this class, but they must be declared as private.

3. Algorithm for Computing a Path to the Destination Cell

Below is a description in pseudocode of an algorithm for looking for a path from the starting cell to the destination. Make sure you understand the algorithm before you implement it.

- Create an empty stack of the class DroneStack.
- Get the starting cell using the methods of the supplied class DroneMap (class described below).
- Push the starting cell into the stack and mark the cell as inStack. You will use methods of the class MapCell to mark a cell.
- **While** the stack is not empty and the destination has not been reached perform these steps:
  - Peek at the top of the stack to get the current cell.
  - If the current cell is the destination cell, then the algorithm exits the loop.
  - If any of the neighbouring cells to the current one has a tower, then pop the top cell from the stack and mark the popped cell as outOfStack.
  - Otherwise, find the best unmarked neighbouring cell (use method nextCell from class Path to do this). If the cell exists, push it into the stack and then mark it as inStack; otherwise, there are no unmarked neighbouring cells, so pop the top cell from the stack and mark it as outOfStack.

4. Running the Program

Your program **must** read the name of the input map file from the command line if you run the program from the terminal, or from the Eclipse configuration if you are using Eclipse (please read the first assignment to remind you how to configure Eclipse so it will be able to find the input map file and all other files that are used in this assignment). From the command line you can run the program with the following command:

```
java Path name_of_map_file
```

where `name_of_map_file` is the name of the file containing the city map. In Eclipse you must write the name of the input file in the Argument tab of Eclipse's configuration as explained in Assignment 1.

You can use the following code fragment to verify that the program was invoked with the correct number of arguments:

```
public class ComputePath {
        public static void main (String[] args) {
                try{
                        if(args.length < 1)
                                throw new IllegalArgumentException(
                                "Provide the name of the file with the input map");
                        String mapFileName = args[0];
                        ...
```

5. Classes Provided

You can download from OWL several java classes that allow your program to display the map on the screen and to see the path that your program has selected. You are encouraged to study the given code so you learn how it works. Below is a description of some of these classes.

5.1 Class *DroneMap*.java

This class represents the map of the city over which the drone will fly. The methods that you might use from this class are the following:

- public DroneMap (String inputFile) throws InvalidMapException, FileNotFoundException, IOException. This method reads the input file and displays the map on the screen. An InvalidMapException is thrown if the inputFile has the wrong format.
- public MapCell getStart(). Returns a MapCell object representing the cell where the UWO store is located.

5.2 Class *MapCell.java*

This class represents the cells of the map. Objects of this class are created inside the class Map when its constructor reads the map file. The methods that you might use from this class are the following:

- public MapCell getNeighbour (int i) throws InvalidNeighbourIndexException. As explained above, each cell of the map has up to six neighbouring cells, indexed from 0 to 5. This method returns either a MapCell object representing the i-th neighbor of the current cell or null if such a neighbor does not exist. Remember that if a cell has fewer than 6 neighbouring cells, these neighbours do not necessarily need to appear at consecutive index values. So, it might be, for example, that this.getNeighbour(0) and this.getNeighbour(3) are null, but this.getNeighbour(i) for all other values of i are not null.
  An InvalidNeighbourIndexException exception is thrown if the value of the parameter i is negative or larger than 5.

- public boolean methods: isTower(), isFree(), isNoFlying(), isThief(), isHighAltitude(), isInitial(), isCustomer(), return true if this MapCell object represents a cell of type tower, free, no-flying, thief, high-altitude, the initial cell where the UWO sore is, or the destination cell where the student's house is, respectively.
- public boolean isMarked() returns true if this MapCell object represents a cell that has been marked as inStack or outOfStack.
- public void markInStack() marks this MapCell object as inStack.
- public void markOutStack() marks this MapCell object as outOfStack.

6. Image Files and Sample Input Files Provided

You are given several image files that are used by the provided java code to display the different kinds of map cells on the screen. You are also given several input map files that you can use to test your program. In Eclipse put all these files inside your project file in the same folder where the src folder is. ***Do not*** put them inside the src folder as Eclipse will not find them there. Please read the instructions in Assignment 1 to remind yourself how to configure Eclipse.

7. Non-Functional Specifications

- **Assignments are to be done individually and must be your own work**. **Software will be used to detect cheating**.
- You must properly document your code by adding comments where appropriate. Add comments at the beginning of your classes indicating who the author of the code is and giving a brief description of the class. Add comments to methods to explain what they do and to instance variables to explain their meaning and/or purpose. Also add comments to explain the meaning of potentially confusing parts of your code.
- Use Java coding conventions and good programming techniques:
    - Use meaningful variable and method names.
    - Use consistent conventions for naming variables, methods, constants, and classes.
- Readability. Use indentation and white spaces in a consistent manner to improve the readability of your code.

8. Submitting your Work

- You **MUST SUBMIT ALL THE JAVA FILES THAT YOU WROTE through the Gradescope link in OWL**.
- **DO NOT** put a ``package" line at the top of your java files.
- **DO NOT** submit a compressed file (.zip, .tar, .gzip, ...); **SUBMIT ONLY** .java files.
- Do not submit your .class files. If you do this and do not submit your .java files, your assignment cannot be marked.
- Submit your assignment on time. Late submissions will receive a penalty as specified in the course outline.

9. Files to submit

- DroneStack.java
- Path.java

Remember **you must do** all the work on your own. **Do not copy** or even look at the work of another student. All submitted code will be run through similarity-detection software.

10. Marking

What You Will Be Marked On:

- Functional specifications:
  - Does the program behave according to specifications?
  - Does it produce the correct output?
  - Are your classes implemented properly?
  - Are you using appropriate data structures?
- Non-functional specifications: as described above.
- The assignment has a total of 20 marks.

11. Marking Rubric

- Program Design and Implementation.
  - DroneStack Class Implementation (2 mark)
  - Path Class Implementation (4 marks)
- Testing. Program produces the correct output for all tests: 12 marks.
- Programming Style: 2 marks
  - Meaningful names for variables and constants: 0.5 mark
  - Code is well designed (simple to follow, no redundant code, no repeated code, no overly complicated code, …) and readable (good indentation): 0.5 mark
  - Code comments: 1 mark