**CS1027: Assignment 2**

**Due: February 27, 11:55pm.**

**Weight: 9%**

**Purpose**

To gain experience with

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

**1. Task**

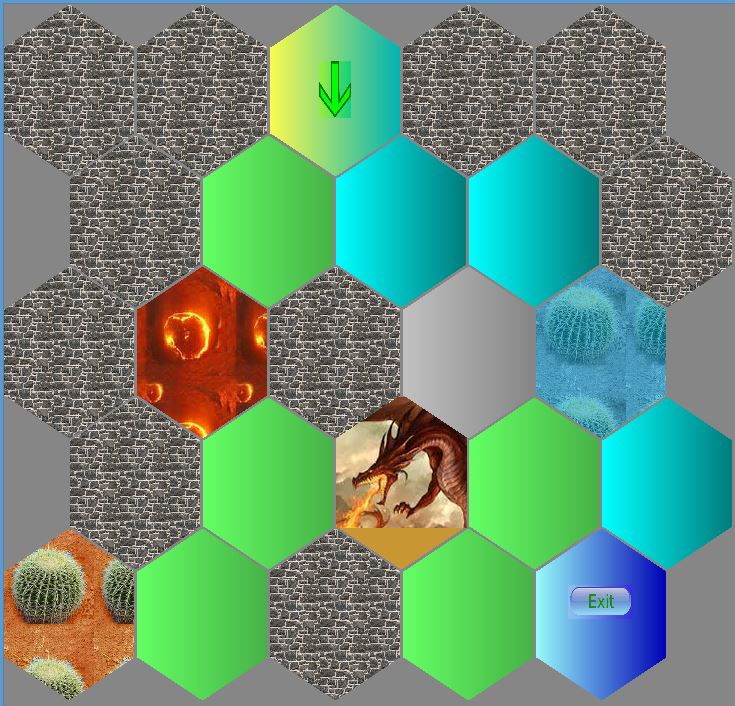
A warrior has been trapped in the dungeon of the Bee Queen. For this assignment you will design and implement a program in Java to help the warrior find a way to exit the dungeon. The dungeon consists of a set of hexagonal shaped chambers. You are given the chamber where the warrior has been initially placed by the Queen. The warrior can move between adjacent chambers and his goal is to reach the exit to the dungeon, if one exists.

There are 6 types of chambers:

* Empty chambers. The warrior can safely walk across these chambers to move to an adjacent one.
* Cacti chambers. The floor of each one of these chambers is full of cacti, the favorite plant of the Queen. The soles of the shoes of the warrior are very thin, so to walk across one of these chambers is a very painful, but non-lethal, ordeal for him.
* Lava chambers. These chambers are shallow lava pools, that provide heat to the dungeon. The warrior has thick skin, so he can swim across these lava pools, but he will be painfully (although not lethally) burned.
* Dragon chambers. Each one of these chambers is a dragon lair. Dragons do not like to be disturbed and they have acute hearing, so the warrior cannot go through these chambers or through any chamber adjacent to a dragon chamber as the dragon’s fire has a long reach. The warrior can enter a dragon chamber or a chamber adjacent to a dragon chamber, but when he realizes that he is in one of these chambers he has to leave it right away and try to find another way to reach the exit.
* Wall chambers. The warrior cannot enter them.
* Exit chamber. This allows the warrior to exit the dungeon.

Initially the warrior is positioned in an empty chamber. When given the choice between going to a free, a cacti, or a lava chamber, the warrior will prefer the free one over the other two and he will prefer a cacti over a lava chamber.

The following figure shows an example of a dungeon. The initial chamber is marked with a green arrow. There is one dragon chamber, a lava chamber and two cacti chambers. The exit is located near the bottom right. The path from the starting chamber to the exit is marked in blue. Note that the gray chamber adjacent to the dragon cannot be part of the solution, so the warrior has to go through a cacti chamber.



Wall chamber

Exit

Free chamber

Cacti chamber

Cacti chamber

Initial chamber

Lava chamber

**2. Classes to Implement**

You need to write a java class called ArrayStack:

*public class ArrayStack<T> implements StackADT<T>*

that implements a stack using an array. You can modify the code provided in the lecture notes to implement this class. This class will have two private instance variables:

* *private T[] stack*. This array will store the data items of the stack.
* *private int top*. This variable will contain the index of the last data item in the array. So, if the stack has 3 data items, then the value of top is 2, as the last data item is in index 2 of the array. **Notice that this is different** from how *top* was defined in the lecture notes. **The initial value for *top* will be -1**.

This class needs to provide the following methods.

* *public ArrayStack()*. Creates an empty stack. The initial capacity of the array used to store the data of the stack is 5.
* *public ArrayStack(int initialCapacity).* Creates an empty stack using an array with the initial capacity given by the value of the parameter.
* *public void push (T dataItem) throws FullStackException*. Adds the *dataItem* to the top of the stack. If the array storing the data items is full, you will increase its capacity as follows:
  + If the capacity of the array is smaller than 20, then the capacity of the array will be increased by a factor of 2.
  + Otherwise, the capacity of the array will increase by 10. So if, for example, the capacity of the array is 30 the new capacity will be 40.
  + If the capacity of the array is increased to more than 1000 (i.e. 1001 or larger) a *FullStackException* exception will be thrown. When creating an object of the class *FullStackException*, a *String* message needs to be passed as parameter to the constructor. We upper bounded 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* in 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().* Returns a String representation of the stack.

You also need to implement a java class called *FindExit*, which will contain the following methods:

* *public FindExit (String filename).* This is the constructor for the class. It receives as input the name of the file containing the description of the dungeon. In the constructor you must create an object of the class *Dungeon* passing as parameter the given input file. This will display the dungeon on the screen. Description of the class *Dungeon* is given below. Some sample input files are provided to you. Read them if you are interested to know the format of these files.
* *public static void main (String[] args).* This method will first create an object of the class *FindExit* using the above constructor. The parameter for the constructor will be the first command line argument of the program, i.e. *args[0]* (see Section 4). Then this method will try to find a way to go from the initial chamber to the exit according to the restrictions specified above. Your algorithm to look for a possible path from the initial chamber to the exit **must use a stack**. Suggestions on how to look for this path are given below. The program must show the chambers selected by the algorithm as it tries to find the exit, so you can visually verify how the program works; see Section 3 and class *Hexagon* in Section 5.
* *private boolean adjacentToDragon(Hexagon chamber).* The parameter is the chamber where the warrior currently is. Class *Hexagon*, described in Section 5, represents the dungeon’s chambers. This method will return *true* if any of the adjacent chambers is a dragon’s lair, and it will return *false* otherwise. Read Section 5 to learn how to find the chambers adjacent to the given chamber.
* *private Hexagon bestChamber(Hexagon chamber)*. The parameter is the chamber where the warrior currently is. This method returns the best chamber for the warrior to move to from the current one. If several unmarked chambers (details about marked and unmarked chambers are given in Sections 3 and 5) are adjacent to the current one, then the method should return one of them in this order:
  + an empty chamber, if one exists. If there are several empty chambers, then the first one (the one with smallest index) is returned. See the description of the class *Hexagon* below to see how neighbouring chambers are indexed.
  + a cacti chamber, if no empty chamber exists
  + a lava chamber, if there is no free or cacti chamber.

If there are no unmarked adjacent chambers the method returns null.

Your program must catch any exceptions that might be thrown. For each exception thrown an appropriate message must be printed. The message must explain what caused the exception to be thrown instead of just a generic message saying that an exception was thrown.

**3. Algorithm for Exploring the Dungeon**

Here is a description in pseudocode of an algorithm for trying to find a path from the starting chamber to the exit. Make sure you understand the algorithm before you implement it.

* Create an empty stack.
* Get the starting chamber from class *Dungeon*, described below.
* Push the starting chamber into the stack and mark the chamber as *pushed*. Read the description of the class *Hexagon* to learn how to mark a chamber.
* Now, ***while*** the stack is not empty, *and the exit* has not been found perform the following steps:
  + Peek at the top of the stack to get the current chamber.
  + If the current chamber is the exit, then the algorithm exits the loop.
  + If any of the neighbouring chambers has a dragon, then pop a chamber from the stack and mark the popped chamber as *popped*.
  + Otherwise, find the best unmarked neighbouring chamber; recall the order in which chambers are preferred. Push this chamber into the stack and then mark it as *pushed*.
  + However, if there are no unmarked neighbouring chambers, pop the top chamber from the stack. Mark the popped chamber as *popped*.

Your program must print a message indicating whether the exit was found or not. If a solution was found the algorithm must also print the number of chambers in the path from the start to the exit. Notice that your algorithm does not need to find the shortest path from the starting chamber to the exit.

**4. Command Line Arguments**

Your program must read the name of the dungeon file from the command line. The command line is what computers use to run programs instead of using graphical windows; all systems still have this functionality available (like through the terminal application on Macs/Linux, or the command prompt (cmd) application on Windows). The user can run the *FindExit* program with the following command from a command line:

java FindExit dungeonFile

If you have ever wondered what the "*String[] args*" thing meant in the *main* method header, this is what it is for: It allows the program to read in the text supplied on the command line. The *args* array of *Strings* will store the text supplied by the user after the java FindExit command.

To read a single argument, we look in *args[0],* but first we want to check that the user has entered an argument. The following code example could be the beginning of your *FindExit.java* file. It will check the length of the *args* array and throw an exception if there is not an argument provided. Then it will store a reference to the provided *String* using the reference variable *dungeonFileName*.

public class FindExit {

public static void main (String[] args) {

try{

if(args.length < 1)

throw new IllegalArgumentException(

"Please provide a file as a command line argument");

String dungeonFileName = args[0];

//...

To get Eclipse to supply a command line argument to your program when it runs, you need to modify the "Run Configurations". Open the "Run -> Run Configurations..." menu item ("Project"->"Run Configurations" on some versions of eclipse). Make sure the "Java Application->FindExit" is the active selection on the left-hand side. Select the "Arguments" tab. Enter the filename and location in the "Program arguments" text box.

**5. Classes Provided**

You are given several java classes that allow your program to display the dungeon on the screen. You are encouraged to study the given code so you learn how it works. Below is a description of some of these classes. Full documentation for the code provided is in the course’s website.

* Class *Dungeon*. This class represents the dungeon. The methods that you might use from this class are the following:
  + *public Dungeon (String inputFile) throws InvalidDungeonCharacterException, FileNotFoundException, IOException.* Reads the input file and displays the dungeon on the screen. An *InvalidDungeonCharacterException* is thrown when the *inputFile* contains an invalid character. Look at the sample input files to learn which characters are allowed.
  + *public Hexagon getStart().* Returns a *Hexagon* object representing the starting chamber.
* Class *Hexagon*. This class represents the chambers of the dungeon. Objects of this class are created inside class *Dungeon* when the dungeon file is read. The methods that you might use form this class are the following:
  + *public Hexagon getNeighbour (int i) throws InvalidNeighbourIndexException*. Each chamber of the dungeon has up to six neighbouring chambers, indexed from 0 to 5. For each value for the index *i*, from 0 to 5, the method might return either a *Hexagon* object representing a chamber or *null*. Note that if a chamber has fewer than 6 neighbouring chambers, these neighbours do not necessarily need to appear at consecutive index values. So, it might be 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: *isDragon*(), *isEmpty*(), *isWall*(), *isLava*(), *isCacti*(), *isExit*(), return true if *this* *Hexagon* object represents a chamber of type dragon, empty, wall, lava, cacti, or exit, respectively.
  + *public boolean isMarked*() returns true if *this* *Hexagon* object represents a chamber that has been marked as pushed or popped.
  + *public void markPushed*() marks *this* *Hexagon* object as pushed.
  + *public void markPopped*() marks *this* *Hexagon* object as popped.

**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 chambers on the screen. You are also given several input dungeon files that you can use to test your program. In Eclipse put all these files inside your project file in the same directory where the default package and the JRE System Library are. ***Do not*** put them in the src folder as Eclipse will not find them there. If your program does not display the dungeon correctly on your monitor, you might need to move these files to another folder, depending on how your installation of Eclipse has been configured.

**7. Extra Credit (5% Bonus)**

Note that the algorithm described above is guaranteed to find a way to reach the exit if there is a way to do so by going through empty, cacti, and/or lava chambers that are not adjacent to a dragon chamber. However, the algorithm might not find a path from the initial chamber to the exit that goes only through empty chambers, even if such a path exists. To get the extra bonus you need to design and implement an algorithm that either

* finds a way to go from the initial chamber to the exit by going only through empty chambers if such a path exists, or if this path does not exist, then
* finds a path from the initial chamber to the exit that goes only through empty and cacti chambers if such a path exists, or if this path does not exist then
* finds a path from the initial chamber to the exit that goes through, empty, cacti, or lava chambers, if such a path exists.

None of the above paths can go through a chamber that is adjacent to a chamber hosting a dragon.

**8. Non-functional Specifications**

1. Assignments are to be done individually and must be your own work. Software will be used to detect cheating.
2. Include comments in your code in javadoc format. Add javadoc comments at the beginning of your classes indicating who the author of the code is and a giving a brief description of the class. Add javadoc comments to methods and instance variables. Read information about javadoc in the second lab for this course.
3. Add comments to explain the meaning of potentially confusing parts of your code.
4. Use Java coding conventions and good programming techniques. **Read the notes about comments, coding conventions and good programming techniques in the first assignment**.

Submit all your .java files to OWL. **DO NOT** put the code inline in the textbox. **DO NOT SUBMIT YOUR *.class* FILES. IF YOU DO THIS, AND DO NOT ATTACH YOUR *.java* FILES, YOU WILL RECEIVE A MARK OF ZERO!**

**9. What You Will Be Marked On**

1. Functional specifications:

* Does the program behave according to specifications?
* Does it run with the test input files provided?
* Are your classes created properly?
* Are you using appropriate data structures?
* Is the output according to specifications?

1. Non-functional specifications: as described above
2. Assignment submission: via OWL assignment submission.