Learning Outcomes

By completing this assignment, you will gain skills in:

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

1 Introduction

Bryan the Bold is on a treasure hunt in the deadly desert in order to become rich. The desert is represented as a grid of cells, consisting of pleasant pathways, glimmering gemstones, and black holes of nothingness. Bryan wants to fill his bag with gems that he can safely reach before returning home. In this assignment you will create a Java program that navigates Bryan through the desert to find as many gems as can fit in his bag before returning to his starting position. The program needs to determine whether the set of pathways in the desert grid can be used to form a path from Bryan to the gems.

You are given a map of the desert, which is divided into rectangular cells to simplify the task of computing the required path. There are different types of map cells:

- An initial map cell, where Bryan is located,
- Map cells where the gems are located,
- Map cells containing black holes, and
- Map cells containing pathways. There are 3 types of pathways:
  - Cross paths. A cross path located in a cell L can be used to interconnect all the neighbouring map cells of L. A cell L has at most 4 neighbouring cells that we will denote as the north, south, east, and west neighbours. The cross path can be used to interconnect all neighbours of L;
  - Vertical paths. A vertical path can be used to connect the north and south neighbours of a map cell; and
  - Horizontal paths. A horizontal path can be used to connect the east and west neighbours of a map cell.

Map cells containing gems interconnect all the neighbouring map cells just like cross paths.

Figure 1 shows an example of a map divided into cells. Each map cell has up to 4 neighbouring cells indexed from 0 to 3. Given a cell, the north neighbouring cell has index 0 and the remaining neighbouring cells are indexed in clockwise order. For example, in Figure 1 the neighbouring cells of cell 8 are indexed from 0 to 3 as follows: neighbour with index 0
is cell 5, neighbour with index 1 is cell 9, neighbour with index 2 is cell 11, and neighbour with index 3 is cell 7 (note that even though there are 4 neighbours, only 2 of them are actually accessible: cell 5 and cell 11, because cell 8 is a vertical path and thus does not connect on the east or west sides). Some cells have fewer than 4 neighbours and the indices of these neighbours might not be consecutive numbers; for example, cell 6 in Figure 1 has 2 neighbours indexed 1 and 2 (no 0 or 3 neighbours).

A path from Bryan (cell number 1 in the figure) to a gem (cell number 7) is the following: 1, 2, 3, 4, 5, 6, 7. A path from Bryan to another gem (cell number 10) is the following: 1, 2, 3, 4, 9, 10. Notice that the two other gems in the figure are inaccessible because of the path types of their neighbouring cells.

Figure 1: A sample of a desert map represented as a grid of cells.
1.1 Valid Paths

When looking for a path the program must satisfy the following conditions:

- The path can go from Bryan, a cross path cell, or a gem cell to the following neighbouring cells:
  - A gem cell,
  - A cross path cell,
  - The north cell or the south cell, if such a cell is a vertical path, or
  - The east cell or the west cell, if such a cell is a horizontal path.

- The path can go from a vertical path cell to the following neighbouring cells:
  - The north cell or the south cell, if such a cell is either Bryan, a cross path cell, a gem cell, or a vertical path cell.

- The path can go from a horizontal path cell to the following neighbouring cells:
  - The east cell or the west cell, if such a cell is either Bryan, a cross path cell, a gem cell, or a horizontal path cell.

If while looking for a path the program finds that from the current cell there are several choices as to which adjacent cell to use to continue the path, your program must select the next cell for the path in the following manner:

- The program prefers a gem cell over the other cells;
- If there is no gem cell adjacent to the current cell, then the program must prefer the cross path cell over the other cells; and
- If there is no cross path cell, then the program chooses the smallest indexed cell that satisfies the conditions described above.

2 Classes to Implement

A description of the classes that you need to implement in this assignment is given below. You can implement more classes, if you want. 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 implements collections. The data structure that you must use for this assignment is an array, as described below.

2.1 ArrayStack.java

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

- `public class ArrayStack<T> implements ArrayStackADT<T>`

You can download `ArrayStackADT.java` from the course’s website. This class will have the following private instance variables:

- `T[] stack`. This array stores the data items of the stack.
- `int top`. This variable stores the position of the last data item in the stack. In the constructor this variable must be initialized to -1, this means the stack is empty. Note that this is different from the way in which the variable `top` is used in the lecture notes.

This class needs to provide the following public methods:
• *ArrayStack().* Creates an empty stack. The default initial capacity of the array used to store the items of the stack is 25.

• *ArrayStack(int initialCapacity).* Creates an empty stack using an array of length equal to the value of the parameter.

• *void push(T dataItem).* Adds *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 100, then the capacity of the array will be increased by a factor of 3.
  - Otherwise, the capacity of the array will increase by 25. So, if, for example, the size of the array is 225 and the array is full, when a new item is added the size of the array will increase to 250.

• *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. If after removing a data item from the stack the number of data items remaining is smaller than one fourth of the length of the array and the length of the array is larger than 40 you need to shrink the size of the array by half; to do this create a new array of size equal to half of the size of the original array and copy the data items there. For example, if the stack is stored in an array of size 100 and it contains 25 data items, after performing a *pop* operation the stack will contain only 24 data items. Since $24 < 100/4$ then the size of the array will be reduced to 50. When creating an *EmptyStackException* an appropriate *String* message must be passed as parameter.

• *T peek() throws EmptyStackException.* Returns the data item at the top of the stack **without** removing it. An *EmptyStackException* is thrown if the stack is empty.

• *boolean isEmpty().* Returns true if the stack is empty and returns false otherwise.

• *int size().* Returns the number of data items in the stack.

• *int length().* Returns the capacity of the array *stack*.

• *String toString().* Returns a *String* representation of the stack of the form:
  
  “Stack: *elem*$_1$, *elem*$_2$, …” where *elem*$_i$ is a *String* representation of the *i*-th element of the stack. If, for example, the stack is stored in an array called *s*, then *elem*$_1$ is *s*[0].toString(), *elem*$_2$ is *s*[1].toString(), and so on.

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

2.2 StartSearch.java

This class will have the following private instance variable:

• *Map desertMap;*

This variable will reference the object representing the desert map where Bryan and the gems are located. This variable must be initialized in the constructor for the class, as described below. You must implement the following methods in this class:

• *StartSearch(String filename).* This is the constructor for the class. It receives as input the name of the file containing the description of the desert map. In this method you must create an object of the class *Map* (described in Section 5) passing as parameter
the given input file; this will display the map on the screen. Some same input files are
also provided in the course’s website. Read them if you want to know the format of
the input files.

• static void main(String[] args). This method will first create an object of the class
StartSearch using the constructor StartSearch(args[0]). When you run the program,
you will pass as command line argument the name of the input file (see Section 4). Your main method then will try to find a path from Bryan to the gems according to
the restrictions specified above. The algorithm that looks for a path from the initial
cell to the destinations must use a stack and it cannot be recursive. Suggestions
on how to look for this path are given in the next section. The code provided to you
will show the path selected by the algorithm as it tries to reach the gem cells, so you
can visually verify if your program works.

• MapCell bestCell(MapCell cell). The parameter is the current cell. This method re-
turns the best cell to continue the path from the current one, as specified in Section
1.1. If several unmarked cells (details about marked and unmarked cells are given in
Sections 3 and 5) are adjacent to the current one and can be selected as part of the
path (as described in Section 1.1), then this method must return one of them in the
following order:
  – A gem cell
  – A cross path cell (if the conditions of Section 1.1 are satisfied). If there are several
    possible cross path cells satisfying the conditions of 1.1 then the one with
    the smallest index is returned. Read the description of the class MapCell below to
    learn how to get the index of a neighbouring cell.
  – A vertical path cell or a horizontal path cell with smallest index that satisfies the
    conditions stated in Section 1.1

If there are no unmarked cells adjacent to the current one that can be used to continue
the path, this method returns null.

Your program must catch any exceptions that are thrown by the provided code. For each
exception caught, an appropriate message must be printed. The message must explain what
caused the exception to be thrown.

You can write more methods in this class, if you want to, but they must be declared as
private.

2.3 EmptyStackException.java

This is the class of exceptions that will be thrown by the methods pop and peek when invoked
on an empty stack.

3 Algorithm for Computing a Path

Below is a description of an algorithm for looking for a path from Bryan to all of the gems.
Make sure you understand the algorithm before you implement it. You do not need to
implement this algorithm. You are encouraged to design your own algorithm, but it must use a stack to keep track of which cells have been processed and it cannot be recursive. You are encouraged to first write a detailed algorithm in pseudocode before implementing it in Java.

- Initialize the number of found gems to 0.
- Create an empty stack.
- Get the start cell (Bryan) using the methods of the supplied class `Map`.
- 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** Bryan’s bag is not full perform the following steps:
  - Peek at the top of the stack to get the current cell.
  - Find the best unmarked neighbouring cell (use method `bestCell` from class `StartSearch` to do this). If one exists:
    - Push the neighbouring cell into the stack and mark it as `inStack`.
    - If the best neighbouring cell is a gem, increase the number of gems found.
  - Otherwise, since there are no unmarked neighbouring cells that can be added to the path, pop the top cell from the stack and mark it as `outOfStack`.
- **While** the stack is not empty perform the following steps:
  - Pop the top cell from the stack and mark it as `outOfStack`.

Your program must print a message indicating how many gems were collected. Note that your algorithm does not need to find the shortest path from Bryan to all of the gems.

4 Command Line Arguments

Your program **must** read the name of the input map file from the command line. You can run the program with the following command:

- `java StartSearch nameOfMapFile`

where `nameOfMapFile` is the name of the file containing the desert map. You can use the following code to verify that the program was invoked with the correct number of arguments:

```java
public class StartSearch {
    public static void main (String[] args) {
        if (args.length < 1) {
            System.out.println("You must provide the name of the input file");
            System.exit(0);
        }
        String mapFileName = args[0];
        ...
    }
}
```

To get Eclipse to supply a command line argument to your program open the "Run -> Run Configurations..." menu item. Make sure that the "Java Application -> StartSearch" is the active selection on the left-hand side. Select the "Arguments" tab. Enter the name of the file for the map in the "Program arguments" text box.
5 Classes Provided

You can download from the course’s webpage several Java classes that allow your program to display the map on the screen. You are encouraged to study the given code to learn how it works. Below is a description of some of these classes.

5.1 Map.java

This class represents the map of the desert including the location of Bryan and the gems. The public methods that you might use from this class are the following:
- `Map(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.
- `MapCell getStart()`. Returns a `MapCell` object representing the cell where Bryan is located.
- `int bagSize()`. Returns the number of gems can that fit in Bryan’s bag.

5.2 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:
- `MapCell neighbourCell(int i) throws InvalidNeighbourIndexException`. As explained in Section 1, each cell of the map has up to four neighbouring cells, indexed from 0 to 3. This method returns either a `MapCell` object representing the i-th neighbour of the current cell or `null` if such a neighbour does not exist. Remember that if a cell has fewer than 4 neighbouring cells, these neighbours do not necessarily need to appear at consecutive index values. So, it might be, for example, that `this.getNeighbour(2)` is `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 3.
- `boolean` methods: `isBlackHole()`, `isCrossPath()`, `isVerticalPath()`, `isHorizontalPath()`, `isBryan()`, `isGem()`, return `true` if this `MapCell` object represents a cell corresponding to a black hole, a cross path, a vertical path, a horizontal path, Bryan, or a gem, respectively.
- `boolean isMarked()` returns `true` if this `MapCell` object represents a cell that has been marked as `inStack` or `outOfStack`.
- `void markInStack()` marks this `MapCell` object as `inStack`.
- `void markOutStack()` marks this `MapCell` object as `outOfStack`.

5.3 Other Classes Provided

`ArrayStackADT.java`, `CellColors.java`, `CellComponent.java`, `InvalidNeighbourIndexException.java`, `CellLayout.java`, `InvalidMapException.java`, `IllegalArgumentException.java`.
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. If your program does not display the map 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 Submission

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 submit your .java files your program cannot be marked. Do not submit a compressed file with your Java classes (.zip, .rar, .gzip, ...). Do not put a "package" command at the top of your Java classes.

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. You can assume that the data in the input files are correct. So no error checking of any type is required.

3. 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 giving a brief description of the class. Add javadoc comments to the methods and instance variables. Read information about javadoc in the second lab for this course.

4. Include comments in each Java file describing key parts of your program. Comments should be grammatically correct, concise, and easy to understand.

5. Use Java coding conventions and good programming techniques. For example:
   (a) Use Java conventions for naming variables and constants.
   (b) Write readable code: good indentation, appropriate white spaces, etc., are required.

6. Make sure your code runs using Eclipse's Java, even if you do not use Eclipse to write your code.
9 Grading Criteria

- Total Marks: [20]
- Functional Specifications:
  - [4] ArrayStack.java
  - [4] StartSearch.java
  - [8.9] 16 Tests
- Non-Functional Specifications:
  - [1] Meaningful variable names, private instance variables
  - [0.6] Code readability and indentation
  - [1.5] Code comments

Assignment files (ArrayStack.java and StartSearch.java are to be submitted to OWL by 11:55pm on October 18th (a Friday).