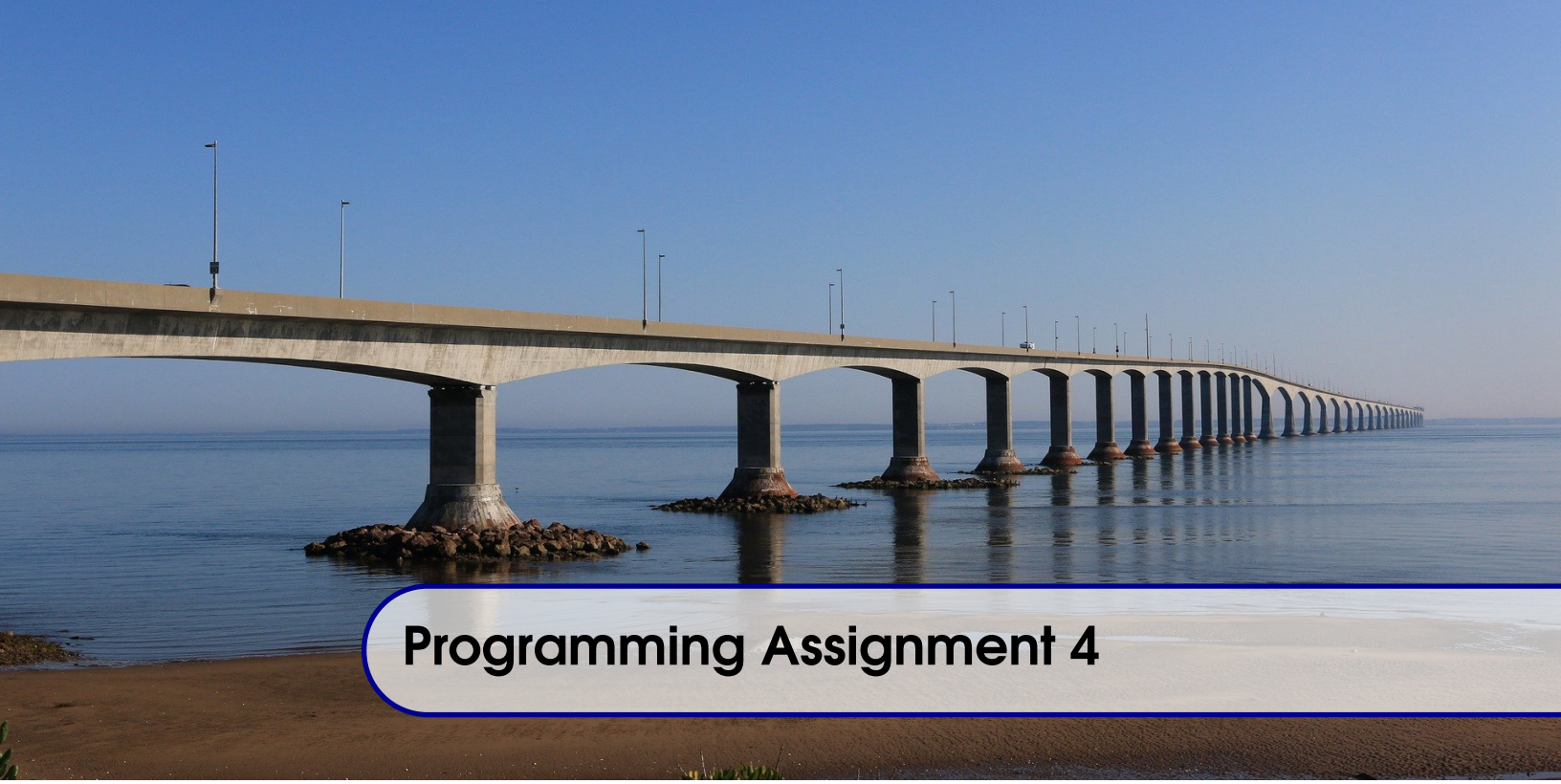

Programming Assignment 4

Assignment Objectives

By completing this assignment you are demonstrating your ability to:

- Create your own classes implementing a Java interface.
- Create your own classes by extending an existing abstract class.
- Read and write data from files.
- Create your own linked structure for representing complex data.
- Design and implemented fairly complex algorithms for manipulating the data.

Scenario: Mazes & Maps

In this and the next assignment we will be working with mazes and maps for representing and navigating through them. In these mazes each room has four walls: east, north, west and south. Each wall may or may not have a gate, and some gates will be locked. You would need a color key to unlock them. Some rooms will also contain such keys.

We will name each room using a string, which could simply be a letter. Different rooms are connected by gates, these are magical gates that have teleport portals in them, and thus, they can arbitrarily connect to each other. Each maze will also have an Entrance and an Exit gate.

There are two types of Mazes:

- Unidirectional Mazes: Portal Gates in this type of mazes are not symmetrical; which means that the south gate from Room A may take you to the north gate of room B, but the the north gate from room B may take you to a different location, e.g. east gate in room C. These mazes are much more difficult to navigate, and thus, we won't be focusing on them.
- Bidirectional Mazes: In these mazes gates have the exact opposite direction either way. This means that if the north gate of room A connects to the south gate of room B, then the south gate of room B will also connect to the north gate of room A. These rooms are easier to navigate

and also easier to describe: **you only need to describe the connection in one direction**. In this assignment we will be working with this type of maze.

Description of a maze

A maze can be described in a text file, with each line providing information of the connection between two rooms' gates, a key contained in the room or the location of the entrance or exit gate. For example, the following line indicates that the west gate of room A connects to the south gate of room B, and vice-versa:

A west <-> B south

This other line indicates that the north gate of room B connects to the south gate of room E, and vice-versa, but also indicates that a BLUE key is required to unlock the gate:

B north <BLUE> E south

A line like this one indicates that the 'Entrance' to the maze is through the south gate of room A.

A south <-> Entrance

And finally, a line like this one, indicates that room B contains a RED key:

B RED key

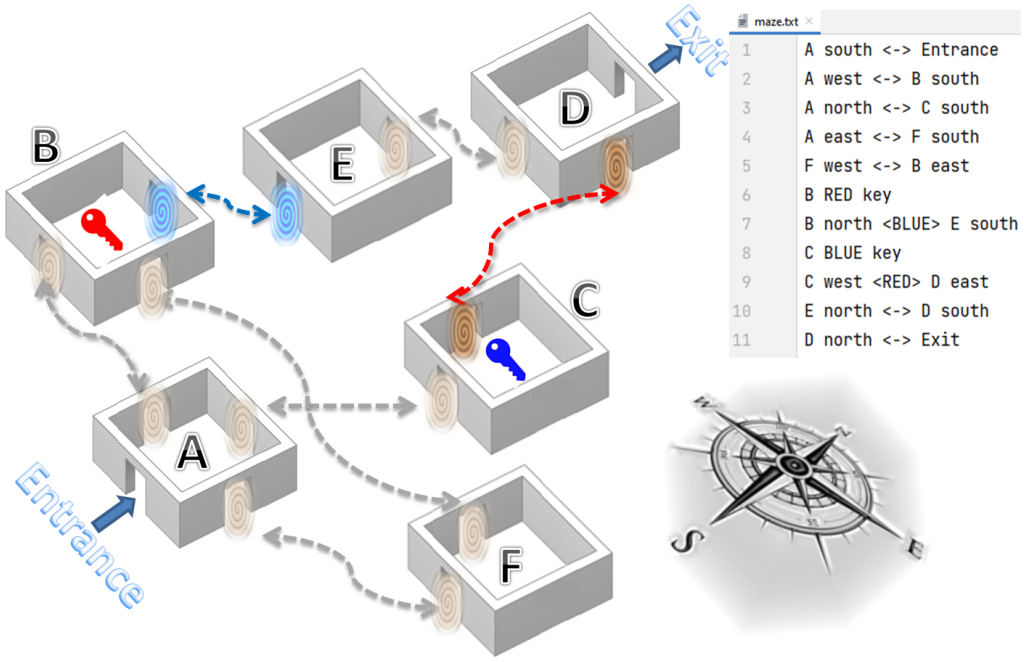

Figure 1: This picture shows a maze and its description in a text file.

The MazeRoom interface and the Action class

You will be provided with an interface for a Maze Room. This interface defines the following methods:

- *public String name()* - returns the name of the room.
- *public String roomKey()* - returns a string in capital letters indicating the color of the key (e.g. "BLUE") or an empty string ("") if no key is present.

MazeRoom interface also extends the generic *Iterable<Action>* interface. This means that for a class to implement the *MazeRoom* interface, it must also implement the *Iterator()* method, which must return an iterator; in this case an iterator of actions. For more information on the Iterable and Iterator interfaces refer to the documentation:
https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html
https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

The *Action* class represents a possible action (specifically a movement) that can be performed in a room. It contains public fields for the gate (String), the room that can be accessed through that gate (MazeRoom) and the required key to unlock the gate (String); empty if none. This class is also provided to you.

The MazeMap hierarchy

There is also a hierarchy of classes for representing maps of different types of mazes. At the top of the hierarchy there is an abstract *MazeMap* class (which you will also get). Concrete classes for BiDirectional and Unidirectional Maze Maps should inherit from the MazeMap class. The MazeMap class contains the following fields and methods:

- *protected MazeRoom entrance* - contains a reference to a 'dummy' entrance node.
- *public void mazeReport()* - prints a report indicating the rooms and keys that are accessible from the entrance.

Problem 1: Save a map to file

Implement an instance method for the MazeMap class:
public boolean saveToFile(String filename) throws Exception
As its name indicates this method will save the map to a file named *filename*, following the same format as shown in Figure 1. Things to take into account when implementing this method:

1. The method must not catch the IOException if it fails to create the file.
2. The method must throw an Exception with the message: "*The maze is empty.*" if there are no rooms in the maze.
3. The method should avoid redundancies in the description of the maze (marks will be deducted), e.g. writing *A west <-> B south* and also *B south <-> A west*.
4. The method is only responsible for writing information about the rooms that are accessible from the entrance.

Figure 2 shows the file used to create a maze map and the file after using the **saveToFile** method to save it back to a different file. You can see that the last line of the original file description was dropped because it is not accessible from the entrance (this should happen as part of the previous assignment). You also notice that the order of the lines in the description changes in the saved file; the order will depend on how you implemented the iterator (also from previous assignment). You should also notice that one line has the order changed: line 9 in the original final and line 10 in the new file. This is okay because this is a bidirectional map.

For Problem 1 you must provide a Driver class, name it 'DriverP1', which will create a maze map object from the provided 'maze.txt' file. The driver will then save the map back to a file; catch any potential exceptions and print the error message to the console.

File used to create the Map

maze.txt

```
1   A south <-> Entrance
2   A west <-> B south
3   A north <-> C south
4   A east <-> F south
5   F west <-> B east
6   B RED key
7   B north <BLUE> E south
8   C BLUE key
9   C west <RED> D east
10  E north <-> D south
11  D north <-> Exit
12  Z south <GREEN> Y north
```

File after saving the map back using ***saveToFile*** method

maze_resaved.txt

```
1   A south <-> Entrance
2   A west <-> B south
3   A north <-> C south
4   A east <-> F south
5   F west <-> B east
6   B RED key
7   B north <BLUE> E south
8   E north <-> D south
9   D north <-> Exit
10  D east <RED> C west
11  C BLUE key
```

Figure 2: This picture shows the file used to create a maze map and the file after using the **saveToFile** method to save it back to a different file.

Problem 2: Find an escape route

Implement an instance method for the MazeMap class:

public boolean noKeysSolution(String filename) throws Exception

This method will save in a file a path from the entrance to the exit of the maze, i.e. a solution for escaping the maze. However, this would be a simplified solution that assumes all the gates are unlocked. This means that if you have a map with locked gates, you can ignore the keys. For example, Figure 3 shows a possible solution to the 'maze.txt' file provided with the assignment (which is the same maze as in Figure 1. The *noKeysSolution* method must throw an exception with the message: "*The maze is empty.*" if there are no rooms in the maze.

maze_solution.txt

```
1   Entrance -> A
2   A -west> B
3   B -north> E
4   E -north> D
5   D -north> Exit
```

Figure 3: An example of a possible solution for the maze shown in Figure 1. This solution ignores the locks.

For Problem 2 you must provide a Driver class, name it 'DriverP2', which will create a maze map object from the provided 'maze.txt' file. The driver will then save the solution to a file; catch any potential exceptions and print the error message to the console.

Academic Integrity Statement

For this, and all assignments in CS2910, you will include a text file in the project folder, at the same folder level as `src` and `data`, named `academic-integrity-statement.txt`. Within that file you should include the following text:

As a student in CS2910 at the University of Prince Edward Island, I am committed:

- To sustain my effort and engagement for the learning in this course.
- To ask questions about the purpose of or criteria for an assessment if I am uncertain.
- To ask questions about the rules surrounding the assessment if I am uncertain.
- To follow the rules for the assessment, including under conditions of no supervision.
- To treat my personal learning as valuable it its own right.

I submit this assessment, acknowledging the above commitments and that I have followed the rules outlined by my instructor and consistent with the Academic Regulation 20.
At the bottom of this text you should put your name, student number and the date of submission.
<STUDENT NAME>
<STUDENT NUMBER>
<ASSIGNMENT DUE DATE>

Please replace the items in angle brackets (<>) with appropriate information for your assignment.
Please note: We have not included this file in the package download. You must add this file yourself with the contents as specified above. If you do not submit this statement you will receive 0 on the assignment.

Submitting your assignment

Your submission will consist of one zip file containing solutions to all problems (see below). This file will be submitted via a link provided on our Moodle page just below the assignment description. This file must be uploaded by 5pm (Charlottetown time) on the due date in order to be accepted. You can submit your solution any number of times prior to the cutoff time (each upload overwrites the previous one). Therefore, you can (and should) practice uploading your solution and verifying that it has arrived correctly.

An assignment received between 5:01pm on the due date (Friday) and 11:59pm on the following day (Saturday) will be accepted as 1 day late and will have a 10% penalty applied. An assignment received between 12:00am and 11:59pm on the second day following the assignment due date (Sunday) will have a 30% penalty applied. All assignments not received by that date will be considered incomplete and will be given a 0.

What's in your zip file?

A zip file is a compressed file that can contain any number of folders and files. Name your zip file using this format.

Example:
Submission file: `asnX_studentnum.zip`
Where `X` is the assignment number between 1 and 4, and `studentnum` is your student ID number.

Your zip file should contain an IntelliJ project and appropriate files for the assignment and your Academic Integrity Statement.

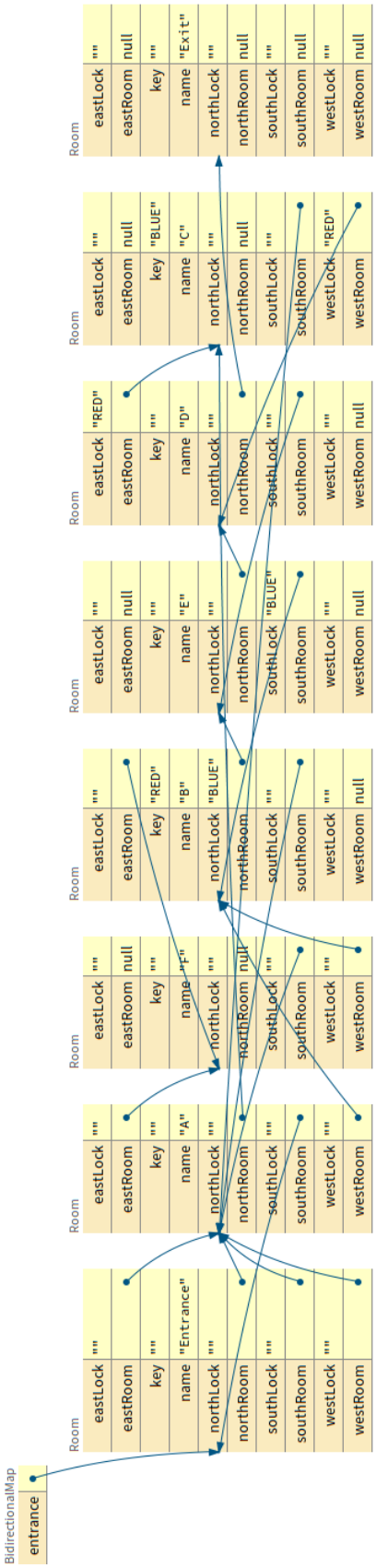

Figure 4: Linked structure visualized by the Java Visualizer tool for the map in Figure 1.