# BASIC PROGRAMMING PROJECT 2017-2018

## aMazeing

Samuel Oswald | Basic Programming (Java) | 02 January 2018

# Introduction

For the class "Basic Programing" at KU Leuven, this maze game was produced as the final assignment. The game imports a text file containing a maze in a predetermined format (which is checked by the program). The player is able to select the maze they would like to play by typing its name into a scanner.

Once the player has entered their name and the maze they would like to play, they are shown a text visualization of the maze, and prompted to make a move. To move, the player must enter a direction using w/a/s/d or z/q/s/d into the scanner. After each move, a string is output to the console detailing whether their move was successful. Along the way, players may also pick up hidden objects which alter the way they interact with the maze. An example of the maze visualization after having made a move, and prior to making another move, is shown to the right.

Once the player reaches the end of the maze, their score (the total moves taken minus points gained by finding trophies) is written to a high score file. The player is given the choice of whether they want to play another maze, in which case a second instance of the run() method is played.

```
d
You have passed through an open space. Nice work!


+---+---+---+---+---+
|   |   |       | E |
+   +   +---+   +   +
|   |           %   |
+   +   +---+   +   +
|           [   |   |
+   +---+   +   +   +
|       |   |       |
+---+---+   +   +   +
|     P     |   |   |
+---+---+---+---+---+
Current moves taken: 1
Current objects:
Make your move!
```

FIGURE 1: VISUALIZATION OF A MAZE, AFTER PLAYER MOVE.

# Classes

In order to ensure the program runs effectively, four classes have been created, with each having a series of methods which allow it to behave as required; as well as to interact with other classes. These classes are RunGame, Maze, Cell, and Player. An overview of each of these classes and its functionality is outlined below. More information is provided within the code.

## Class: RunGame

The RunGame class is in a separate package to the other classes. It includes the main method used to run the program, and prompts the player through the various stages of the game via a Scanner. The three methods of the class are run, highScoreWriter, and the main method. It is in a separate package to the other three classes (maze,

void run() will first create a scanner using a try argument. Prior to a player input, the scanner will prompt whether it requires a maze name, a player name, a movement direction, or if the player would like to play again. After receiving player inputs for each of these, the run() method will then call methods from the maze or player class in order to

move through the game. It uses a while loop while the maze is running to continually call methods from the maze class while it remains unsolved, and also uses get methods from the player class in order to keep the user updated on the current status with regards to items possessed and moves taken. Following the end of the while loop (i.e. the maze is solved), it will call the highScoreWriter method and prompt the player whether they would like to restart the game. Should the player select 'yes' to this prompt, a second instance of run() will be commenced from within the method, otherwise the program will end.

The void highScoreWriter(String, String, Int) method is called from the run() method, and will search for whether a high score file exists. If it does not exist, it will be created. This file is then opened as a bufferedWriter, and a new line(using bufferedWriter.newline() and bufferredWriter.append() methods) containing the players name, the maze name, and the number of moves taken is appended.

The main method first outputs information regarding the game and how to play it to the console, then runs the run() method.

## Class: Maze

The maze class is used to organize the logic of the maze and the relationships between the various cells that make it up, as well as the player that moves through it. Methods include a constructor, as well showMaze, writeLine, findCell, validMove, playerMove and confirmMove methods.

Using an input file, the constructor will find the maximum dimensions of the maze based on lines representing individual cells within the file. These files are required to be of a certain format, with a heading string indicating whether the file confirms to this format. The constructor will throw exceptions if the file format is incorrect, or if the file is not found (however this shouldn't be the case as the files existence is checked previously in the runGame class). If the file has a valid format, the maze will create an array of cells of size maxRow * maxCol (derived from the final line in the input file), and use a for loop to loop through the input file lines and use the Cell class constructor to create a new cell object for each line.

The void showMaze() method can be used on a constructed maze, and will itself call the void writeLine(int, boolean) method for each row in the maze, inputting the current row number and whether this row is the uppermost line. The writeLine method will then parse through the cells on each row using a for loop and call the drawWall or drawObject methods from the related cell object, which will then output a string to the console. It additionally checks for maze logic, and will return exceptions if there are problems with the maze(i.e. edges of the maze which aren't bounded by walls). As cell walls are connected and only need to be drawn once, each cell except for edge cells will only have their south and east walls drawn. This could create problems if two cells don't correctly match up, with potential conflicts being checked through the later validMove method.

int[] findCell(String) loops through every cell in the maze in order to find a certain object, which is input into the method. This is mainly applicable for finding the start of the maze, where the player begins the game.

The Boolean validMove(String) method is called by the runGame run() method, and will check that the string entered prior to a player move corresponds to a movement direction. If it returns true, the run() method will then run void Maze.playerMove(String, Player). playerMove accepts a string containing a direction, as well as the active player as inputs. It retrieves the players current position in the maze, and then sets the cell with this row and column value as the oldCell, and the adjacent cell in direction of the string as the newCell. It takes the wall values of the new and old cell, and checks firstly that these are equal (otherwise throwing an exception as the file format is incorrect), and then assesses whether the player is capable of moving between these cells using the Boolean confirmMove((String wall, boolean, boolean key, Cell, Cell , String, String) method. If the player is capable of moving between them, both of the cells current objects are updated (with the player.takeObject() method being invoked on the newCell), as is the players movesTaken and position in the maze.

## Class: Cell

The cell class contains information pertaining to each individual cell within the maze, namely, the wall properties and object properties in the cell. Methods include checkValue, drawWall, drawObject, updateObject, breakWall and getObject.

The constructor takes a single line from the input file, which is parsed via the Maze class. The constructor checks that the values for each wall and object are included in the list of possible values via a for loop over arrays containing possible values (defined at the top of the Class) via the Boolean checkValue(String, String, String, String, String) method. Any illegal attributes will throw an illegal argument exception.

The String drawWall(String) and String drawObject() methods return strings once the showMaze method is called from the Maze object which contains the cells. These strings when combined through this previous method will define how the maze is displayed to the player. The drawWall method additionally will call on the private drawHoriType(String) and private drawVertType(String) methods, depending on whether a vertical or horizontal line in the cell is being drawn.

void updateObject(String) and void breakWall(String) methods will update the cells upon user interaction. Generally, as a player enters a cell, the String getObject() method will be called by Maze.playerMove(), and if this returns a value other than no, the player will take the object within the cell, and in turn this method will be required to remove the object from the cell. breakWall is used when a player interacts with a wall which they have the item necessary to get through it, either the key or hammer. This will alter the visualization of the maze in the future.

## Class: Player

The player class is created for each maze, and represents the decisions taken by the player as they attempt to solve the maze. Attributes include name, current position in the maze, moves taken in the maze, and what objects it possesses (represented as Boolean values). Methods include takeObject, getPosition, getName, getMoves, getHammer, getKey, getSolved, and setNewPosition. Data about the player is private, and can only be accessed via the methods of the class.

The constructor is called from the RunGame.run() method, following the construction of the maze. It's inputs are the user-defined player name, as well as an initial position corresponding to the "start" cell of the maze. Aside from this, other variables relating to objects possessed are set to "false", whilst movesTaken is set to 0.

The void takeObject(String) will alter one of the Boolean values when the method is called. The string will determine which object is altered, or default to no change.

The Various get methods will return the attributes of the player. They may be used in other Maze methods which need to know whether objects are possessed (i.e. to break walls) or when the RunGame class is updating the console output or high score table.

void setNewPosition(String) will alter either the row or column value of the player position depending on the String input. Additionally, it will increase movesTaken by 1.

## Interfaces: PlayerMethods, CellMethods, MazeMethods

Interfaces are provided for each class in order to create the framework by which polymorphism may be achieved if further extension to the program is made. For example, different types of players or more advanced types of cells may be implemented which will require common functionality with the Maze. The methods provided in the interfaces are a guide for the key methods to create for each class.

# Relationships

There are numerous relationships between the classes through various method calls between them. The most obvious relationship is between the Cell and Maze class, whereby each is dependent on the other to exist logically. A maze provides both the constructor for the Cell class, and is reliant on these cells existing to function properly. When a user interacts with individual cells, it is through a Maze object. Objects of the Player Class are similarly connected to individual cells through the Maze class. For example, the position of the player in the maze is recorded both in the attributes of the Player object, as well as in the cell with corresponding position in the maze. The Maze object will invoke methods in both to ensure that they match. The Maze object additionally hides information from public users, as it invokes methods which are only available to classes within the package on Player and Cell objects.

The figure below demonstrates how classes interact within the program:

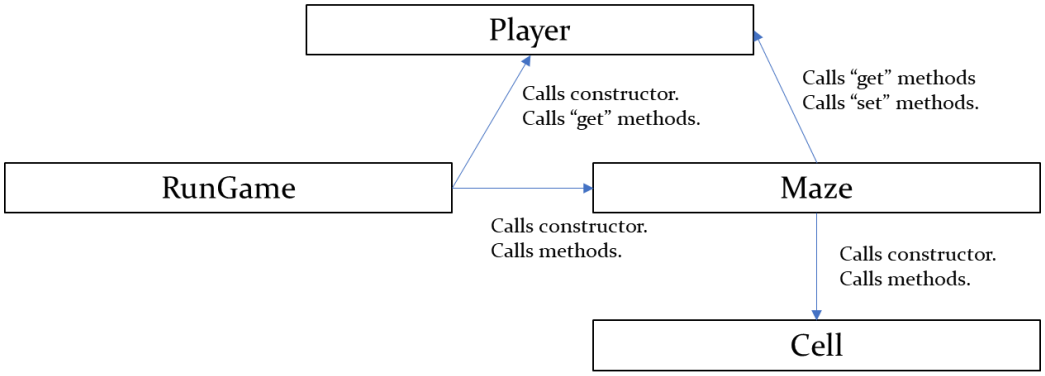


FIGURE 2: RELATIONSHPS BETWEEN CLASSES.

# Conclusions/Reflections

This program satisfies the criteria outlined for the assignment, allowing users to load and play through a maze program, with the results being saved to a high score table. There are several aspects of the program which for which it is particularly strong, but also some drawbacks to the approach taken. An overview of some strengths and weaknesses is shown below, to conclude this report.

## Strengths

1. The flow from the start of the program to the end is sensical for the user, and, assuming the file format is correct, will not crash. Numerous exception catches have been implemented, as well as if/else statements which allow for errors to be detected and avoided. An example of this is at the start of the game, where if a file is not found, it will prompt the user to enter a different maze name, avoiding a File Not Found Exception, which would otherwise occur.
2. The various classes allow for extensibility to the program in the future. For example, the maze class is capable of handling multiple players in the event of, for example, adding a multiplayer option. This would involve setting up an option to add more than one player in the RunGame.run() method, then invoke separate calls to the maze for each player.
3. The organization of the code is done in a way which should make it easily understandable to someone viewing/working on the code in the future. Various naming styles are implemented, and comment blocks are included to explain how various aspects of the code work. Use of whitespace additionally makes the code easily readable. Future extension or improvement of the code can thus be done more easily than if naming conventions weren't used or methods were not adequately explained.

## Weaknesses

1. The interfaces provide a framework for future extensibility of the program, however in its present state OOP features such as inheritance and polymorphism

are not as well integrated. Each class implements an interface, however these interfaces at present only apply to that class. It was decided during the design of the program that the functionality of each class was different enough that no common interfaces could be applied to numerous classes. Additionally, whilst the implementation of these interfaces could provide a framework for future extensibility, it may be easier to also include an abstract class for various types of mazes, cells, or players, so that methods do not need to be redefined.

2. The interface for movement can feel clunky. It would be more user-friendly if when a player moves they would only need to press the movement button in a direction, as opposed to entering a single letter and pressing enter each time. This functionality is possible to implement by using a key listener, however documentation of this possible method appears to be oriented towards GUI-development.
3. Whilst the user experience is clean, there could be greater options available for whenever the user is prompted to input, such as asking for “help”, “restart” the game, “save” the game or further extensions. At present, only the specific action required for that point is allowed, whilst these would ideally be possible at any stage during the use of the program. This could be done by implementing a new Class which administers the possible options for input into the scanner at any one point, which could possibly be customized by the user. In effect, the scanner would allow inputs to match a number of arrays of possible strings, with the potential arrays changing depending on the position within the run() method of the scanner.