CM10227 Coursework 2
Dungeon of Doom (Java)

Dept. of Computer Science
University of Bath

1 Introduction

The assessment of this unit consists of two courseworks. This document provides the specification for the second coursework: **Dungeon of Doom** in **Java**.

You can use any Integrated Development Environment (IDE) for the development of your code, but your code must run on Java 17 (the default version used on repl.it and on the University's linux.bath server) without requiring the installation of additional libraries, modules or other programs. Standard packages included with Java (e.g. `java.util`) are fine.

Questions regarding the coursework can be asked in the lectures or labs, sent to the programming1@lists.bath.ac.uk mailing list, or asked on the Moodle Q&A forum.

2 Learning Objectives

At the end of this coursework you will be able to design, write and document a medium-sized program using appropriate object oriented software techniques.

3 Dungeon of Doom

For your first coursework, we asked you to investigate and replicate a piece of code that we supplied (SRPN). In this second piece of coursework you will need to write code to meet a specification.

Your assignment is to design and write a program that allows a human player to play the game 'Dungeon of Doom', described below. You must also include documentation (either commenting or Javadoc) for your classes and methods, and a README file that states how you have considered object-oriented design principles in your implementation (maximum 200 words).

For additional marks you may implement a computer-controlled player ('bot') to play against the human player.

3.1 Game overview

The *Dungeon of Doom* is played on a rectangular grid, which serves as the game's board. A human player, acting as a brave fortune-hunter, can move and pick up gold. The goal is to get enough gold to meet a win condition and then exit the dungeon. A bot player, acting as a villain, is trying to catch our hero. The game is played in turns. On each player's turn, that player (human or bot) sends a command and (if the command is successful) an action takes place. A full list of the available commands, the *game protocol*, is on the next page.

The game ends either:

- when the human player has collected enough gold **and** calls the EXIT command on the exit square or
- when the bot catches (moves onto the same square as) the human player.

3.1.1 Board representation

The dungeon is made up of square tiles. Each tile can be:

1. **Player:** This tile represents a human player. It is displayed as the letter *P*.
2. **Bot:** This tile represents a computer-controlled player, or 'bot'. It is displayed as the letter *B*.
3. **Empty Floor:** Allows a player to walk over it, some may also contain items such as gold. If empty, it is displayed as a *period (.)*
4. **Gold:** A special floor tile which allows a player to walk over it and pickup the gold in it. If (and only if) the human player pickups the gold, then the tile is converted into an *empty floor* tile. It is displayed as the letter *G*.
5. **Exit:** A special floor tile that the human player can use to exit the dungeon and win the game by using the QUIT command. It is displayed as the letter *E*.
6. **Wall:** Blocks a player from moving through it. It is displayed as a *hash sign(#)*

3.2 Set-up

The Dungeon of Doom is loaded from a text file. Additional marks will be awarded for allowing the user to select a map file at launch - this may be the choice of multiple map files or specifying a file location. The human player starts the game with no gold, and at a random location within the dungeon. This position must not contain the computer-controlled player or any gold, but it may be an exit tile. The computer-controlled player also starts at a random location within the dungeon. The players must not be placed inside a wall.

3.3 Game Protocol commands

Your software must allow players to use commands from the 'game protocol' (below) on their turns and see the response to those commands. For the human player, these commands will be entered through the command line. Commands may be upper or lower case. Note: you should not include <and >, these denote where a value should be inserted.

HELLO

The response displays the total amount of gold required for the player to be eligible to win. This number should not decrease as gold as collected. Example format: *Gold to win: <number>*

GOLD

The response displays the current gold owned. Example format: *Gold owned: <number>*

PICKUP

Picks up the gold on the player's current location. The response is `Success` and the amount of gold that the player has **after** picking up the gold on the square. If there is no gold on the square, the response is `Fail` and the amount of gold that the player had before attempting PICKUP. Example format is: *Success. Gold owned: <number>*

MOVE <direction>

Moves the player one square in the indicated direction. The direction must be either *N, S, E or W*. For example, `MOVE S`. Players cannot move into walls. The response should be either `Success` or `Fail` depending on whether the move was successful or not.

LOOK

The response is a 5x5 grid, showing the map around the player. The grid should show walls, empty tiles, gold, exits, and players with the relevant character or symbol. The calling player must be shown at the center of the grid with a *P* (human) or *B* (bot). Visible areas **outside of the map** should be shown as a wall ('#').

```
Example 1      Example 2
###..          #####
#.#.E          #####
..P..          ..P##
#....          ...##
#..G.          B..##
```

The players must not be able to view the map other than by using the LOOK command.

QUIT

Quits the game. If the player is standing on the exit tile *E* and owns enough gold to win, the response is `WIN`, followed by an optional winning message. Otherwise, the response is `LOSE` and quits the game, losing all progress.

All commands take up a player's turn, regardless of whether they were successful or not. Once a command has been entered, the response should be printed and the turn is over.

3.4 Advance Feature: Bot

For extra marks, you can choose to implement a basic computer-controlled player or 'bot' to compete against the human player. It is suggested that this be implemented in a class called BotPlayer, which returns commands to the GameLogic in the same way that HumanPlayer does. How the bot determines which commands to use on its turn is up to you:

- A minimum implementation will involve the bot randomly moving around the map.
- A more advanced implementation will have the bot looking for and attempting to chase the human player.
- You can also choose to make your bot a **Looter**! Our villain is no longer content to just chase the hero, they want to steal the gold too! The **Looter** bot can pickup gold using the PICKUP command. If it gathers enough gold to win, it can go to an exit tile and leave with the EXIT command. It can still win by catching the human player.

As it is a player, the bot should only be able to view the map through the LOOK command, and should only use commands on its turn.

3.5 Code Specifications

Some basic code has been supplied on Moodle to get you started with this coursework; you are free to use and extend this code as you wish, or start over. The code provides a definition of three classes and some method signatures within these classes.

Below are the classes and some **suggested** implementations to accompany the method signatures:

1. GameLogic: GameLogic includes a *main* method to run the game. This main method creates an instance of the GameLogic class from which to run the game. GameLogic creates a Map and any players, then retrieves commands from the human or computer players. Appropriate accessors are provided to describe the state of the game. GameLogic also tracks which player's turn it is.

2. Map: The Map class holds a reference to the current state of the map (in a 2D char array), the name of the map, and how much gold is required for the human player to win. The code provided includes a default map that is created with the default constructor. This code could be extended to read in a map, name and gold amount from a file (provided on Moodle).

3. HumanPlayer: HumanPlayer prompts the user to input commands on the command line, returns these commands to GameLogic, and display results on the command line.

You may create any other classes, superclasses or otherwise in your implementation. Note that you should create another class if you choose to implement the computer-controlled player, as described in **Advanced Feature: Bot**.

In this coursework, you should write code (which may extend the provided code) to implement Dungeon of Doom. A successful implementation is one where the game can be started, then commands can be entered to play until the game ends. The game ends when the player collects enough gold to meet the win condition and exits the dungeon, or when the human player is caught by the bot. Your code should be able to read in the example map provided on Moodle, but other maps will be used when marking.

You may wish to write your own maps in the testing of your game.

We value clean code and good practices. You will be marked on the quality of your code and commenting.

4 Submission

You must submit a single zip file, with the following in the root of the zip:

1. Your source (`.java`) files required to compile and run Dungeon of Doom. Do not include packages or other subfolders. Do not include compiled (`.class`) files. If you are using an IDE to develop your code, you must extract the source files – do not include the entire project structure or version control files. Penalties will be applied for failing to follow submission instructions.

2. A README.txt file.

3. Any Javadoc files if you have chosen to generate them.

Your zip file must be named `dod-username.zip`

Failure to follow the submission specifications, by providing unneeded files or not following the .zip structure specified, will result in a penalty being applied to the final mark.

Your .zip file must be submitted to the Moodle assignment page by the submission deadline shown. Submissions received after this deadline will be capped at 40% if received within 5 working days. Any submissions received after 5 working days will be marked at 0%. If you have a valid reason for an extension, you must submit an extension request through your Director of Studies – unit leaders cannot grant extensions.

You must leave yourself time to download your file from Moodle, extract it, and check that you have attached the correct file, with the content that you want to be marked. **You** are responsible for checking that you are submitting the correct material to the correct assignment.

Please check Moodle for the submission deadline.

5 Assessment

5.1 Conditions

The coursework will be conducted individually. Attention is drawn to the University rules on plagiarism. While software reuse (with correct referencing of the source) is permitted, we will only be able to assess your contribution.

5.2 Marking Scheme

Below is a breakdown of marks, with descriptions on how each criteria is met.

Criteria	Max Score	Description
Core Specifications		
Game functionality satisfying requirements	max 40	Code satisfies the specification to play the core game as a human player.
Code quality and formatting.	max 25	Code is written cleanly and simply while implementing object-oriented programming techniques.
Commenting	max 10	Code is consistently commented through comments or Javadoc.
Documentation	max 5	README.txt file describes how the code implements object oriented principles
Computer-controlled player		
Basic implementation of a bot	max 5	Bot moves around the map.
Looter bot	max 5	Bot can pick up gold and meet the gold win condition
Further implementation of a bot	max 10	Bot attempts to find and chase the human player.

5.3 Grades

Marks and feedback will be made available within 3 working weeks of the submission.

The following is a guideline for marks:

- **Above 40%** – Pass.
- **50-69%** – Very good. (2:2-2:1 range)
- **70%+** – Outstanding. (1st range)