# CM10227 Coursework 2
# Dungeon of Doom (Java)

November 18, 2019

## 1 Introduction

The coursework of this unit consists of three parts: the lab sheets and two larger courseworks. This document provides the specification for the second large 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 when we use the command-line on Linux.Bath without requiring the installation of libraries, modules or other programs**.

Questions regarding the coursework can be posted on the Moodle Forum, the Slack page, and the programming1@lists.bath.ac.uk mailing list. Tutor support is available in the lab sessions.

## 2 Learning Objectives

At the end of this part of the 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 design and write a program that allows a single human player to play the game 'Dungeon of Doom', described below.

You must also include a README file describing how to play your game, with some comments about your implementation, as well as a set of Javadoc HTML files generated through the *javadoc* command.

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 available below.

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 *full stop (.)*
4. **Gold:** A special floor tile which allows a player to walk over it and pickup the gold in it. 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.

HELLO

Response is a string, displaying the total amount of gold required for the human player to be eligible to win. This number should not decrease as gold is collected. The format of the result is: *Gold to win: <number>*

GOLD

Response is a string, displaying the current gold owned. The format of the result is: *Gold owned: <number>*

MOVE <direction>

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

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. The format is: *SUCCESS. Gold owned: <number>*

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 human player must be shown at the center of the grid with a *P*. Visible areas **outside of the map** should be shown as a wall ('#').

```
###..          #####
#.#.E          #####
..P..          ..P##
#....          ...##
#..G.          B..##
```

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 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 creates 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 human 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 and share 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.**

### 3.5 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.

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. The bot does not need to pickup gold, as its goal is only to catch the human player.

## 4 Marking Scheme

Below is a breakdown of marks, with descriptions on how each criteria is met. More detail will be provided for the Code Review.

## 4.1 Marks Table

| Criteria | Max Score | Description |
|---|---|---|
| Core Specifications | | |
| Functionality satisfying requirements | max 45 | Code satisfied the specification to play the core game as a human player. |
| Code quality, commenting and formatting. | 30 | Code is clearly written and as simple as possible, implements object-oriented programming techniques, and is consistently commented. |
| Documentation | max 5 | The Readme.txt submitted contains useful implementation details and and how to play instructions. Complete Javadoc is provided for all classes. |
| Computer-controlled player | | |
| Basic implementation of a bot. | max 5 | Bot moves around the map. |
| Further implementation of a bot. | max 10 | Bot attempts to find and chase the human player. |
| Code quality, commenting and formatting | max 5 | Code related to the bot uses clean code practices, object-oriented programming techniques, and is consistently commented. |

## 4.2 Indicative Marks

### 4.2.1 Approx 40%:

1. Code satisfied the specification to play the core game as a human-player.
2. Code is properly formatted and commented.
3. Very sparse Readme provided. Javadoc is missing or incomplete.

### 4.2.2 Approx 60%:

1. Meets all the criteria of a 40% pass.
2. Code architecture follows Object-Oriented Principles.
3. Readme contains useful information on the running of your game, plus a brief description of its implementation. Javadoc files are included for all classes.

#### 4.2.3 80% plus:

1. Meets all the criteria of a 60% pass.
2. Bot is implemented.
3. Code quality and commenting is of extremely high quality throughout.
4. As in 4.2.2.

## 5 Submission

You should upload a zip file to Moodle by **10am on 9th December 2019**. The name of the zip file should be in the form:

CW2-<username>.zip
e.g. CW2-ab123.zip

The zip file must contain:

1. Your **source** code and any resources files needed. You should not include packages or other subfolders. **No compiled code or unnecessary files**, e.g. no version control files, should be included.
2. A Readme.txt file containing an introduction to your game, how to run the game, and a few words about your code.
3. Javadoc .html files for all classes in your code.

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