# Java Assignment Report


**Name:** Sun Rong

**School:** University of Aberdeen & South China Normal University


**Major:** Artificial Intelligence

**Grade:** 2022

**Date:** December 10th, 2023

**GitHub:** https://github.com/GarlicToT/Java_Assignment

# Content

# 1. Introduction

## 1.1 Overall design

This is a text-based role-playing game. In this game, players control a character (Player) and interact with monsters (Monster) on the map. The primary objective of the game is to eliminate all monsters in each round through strategic movements and potential attacks, leading to victory. Conversely, if the player's character is defeated by monsters, the game declares a loss. Regarding victory conditions, players must successfully eliminate all monsters on the map to win. If all monsters are defeated, the player achieves victory.

## 1.2 Challenges during development

In the development of this program, I encountered the following challenges:

- When implementing hurtCharacter() method in the Monster class, I made a mistake in writing the statement that determined there was no successful defense as : if (!successfulDefense()) {…}, but actually it should be if (!**character.**successfulDefense()) {…}. It took me two days to debug before I realized what was wrong. Small errors like this took me a really long time to debug!

- When implementing hasOccupiedCharacter() method in GameLogic class, I made it possible to use characrer type and location information to decide whether to attack players/attack monsters/prevent monsters from moving to places occupied by other monsters/characters moving. Achieving so many things in one method was really challenging for me.

# 2. The Development Process

## 2.1 Task 1: Implement the basic methods for the characters

2.1.1) In order to record the code iteration and task process, I first used git function to import the code into GitHub repository, then imported it to the local, and successfully found the **uoa.assignment.character** package in VS Code.

2.1.2) Modify the constructor of **GameCharacter** class to make the field **name** be populated by the parameter value representing name of the character. To achieve this, I have:

- Initialized the character with a name.
- Set the public method **sayName()** to return the name of the character.

2.1.3) Modify the **GameCharacter** class to make the private integer variable **health** be initialized to value 100 by default when the instance is created. To achieve this, I have:

- Initialized health to value 100 by default.
- Used methods getHealth() and setHealth() to provide a way to access and modify the health attribute of a game character object. The method **getHealth()** retrieves the current health value, and the method **setHealth()** allows updating the health with a new value.

2.1.4) Modify the **Monster** class which implements the abstract methods from the GameCharacter class. In Monster.java, I have modified the following methods:

- The method **hurtCharacter()** in Monster class inflicts damage on another GameCharacter object passed as a parameter. It removes 50 health points if the character did not defend successfully.

- The method **successfulDefense()** in Monster class generates a random number (0 or 1) using java.util.Random package to model a 50/50 chance for success or failure in a defense scenario.

2.1.5) Modify the **Player** class which implements the abstract methods inherited from the GameCharacter class. In Player.java, I have modified the following methods:

- The method **hurtCharacter()** in Player class is responsible for causing damage to another GameCharacter object passed as a parameter. If the defense fails, it reduces the health of the targeted character by 20 points. The way to achieve is similar to which used in Monster.java.
- The method **successfulDefense()** in Player class utilizes a random number generator to simulate a 30% chance of unsuccessful defense and a 70% chance of successful defense for a game character.

After completing the above work, Task 1 now implements Git integration, character class enhancements with name and health management, and the introduction of defense mechanisms in the Monster and Player classes with random outcomes. These modifications collectively provide the groundwork for character interactions, health management, and defense mechanisms in the Java project.

## 2.2 Task 2: Implement the map for the playing area

2.2.1) Modify the **Map** class which holds four instances of GameCharacter and adjust the constructor to create a 2D array using the provided height and width parameters. To achieve this, I have:

- Defined a constructor to set up the Map class by creating a 2D array for **layout**, populating it with full stops, and initializing an array to store four different GameCharacter objects, including instances of both Player and Monster.

2.2.2) Create method **initialiseArray()** to populate the 2D array and method **printLayout()** to print the 2D layout array to the console in Map class. To achieve these, I have:

- filled the 2D array with full stop characters.
- Used the method **printLayout()** to print the 2D layout array to the console.

2.2.3) Create instances that one for class **Player** and three for class **Monster** every time a map is created. I have realized the following

functions:

- In the Map constructor, I created an array of **GameCharacter** objects named characters, instantiated one Player object and three Monster objects and stored them in the characters array. The player is stored in position 0, and the monsters are placed in positions 1, 2, and 3 within the array.
- I then set the initial positions of these characters on the map by assigning values to their **row** and **column** attributes.
- I placed the characters in the 2D **layout** array based on their positions. The getCharacterSymbol() method is used to determine whether to represent a character as "*" for the player or "%" for monsters.

2.2.4) Store the positions of monsters and the player in the **layout** array. After implementing the Map class, the initialized array look as follows:

**% . %**
**. . .**
**% . *****

After completing the above work, Task 2 now involved enhancing the Map class to initialize a 2D array for the layout, create instances of Player and Monster characters, set their positions, and store these positions in the layout array. Two methods, **initialiseArray()** and **printLayout()**, were added to manage array population and console printing

## 2.3 Task 3: Implement the basic game functionality

2.3.1) Update the **Game** class constructor to create a **Map** object using **height** and **width** parameters obtained from the program's command-line arguments. The initial map layout is printed to the console during game initialization within the **main()** method.

2.3.2) Extend the while loop in the **RunGame** class. I have:

- Used a Scanner to continuously read user console input until a specified condition for ending the game is met.
- Printed the number of the round at the beginning of each iteration.

2.3.3) Modify the class **GameLogic** and use **moveCharacter()** method to ensure the validity of the **input** direction for a given **character**.

- If the input is "up," "down," "left," or "right," it checks if there is a monster in the target direction using the hasMonster() method.

- If a monster is present, it prints a message and prevents the character from moving.
- Otherwise, it moves the character in the specified direction. If the input is invalid, it prompts the player to use only the specified keywords.

2.3.4) Implement the private static methods **moveRight()**, **moveLeft()**, **moveUp()**, and **moveDown()** within the **GameLogic** class to handle the character movement in different directions. These methods update the character's position in the object instance and the **layout** array of the game map.
I then utilized these methods in the **moveCharacter()** method, allowing characters to move according to user input, with proper validation and handling of invalid directions.

After completing the above work, Task 3 now is enhanced the Game class to create a Map with command-line arguments, implemented continuous user input in RunGame, and ensured valid character movements using GameLogic. The code now prints the initial map layout, handles input validation, and updates character positions in both the object instance and game map layout during each round.
After executing the moveUp() method, the output in Round 1 is as follows:

**% . . . . %**
**. . . . . .**
**% . . . . ***
**Round 1**

## 2.4 Task 4: Implement the game controls and basic logic

2.4.1) Update the moveRight(), moveLeft(), moveUp(), and moveDown() methods in the GameLogic class to print specific messages if a character attempts to move into a wall. The messages are like "You can't go right/left/up/down. You lose a move." to indicate the invalid move and the loss of a move.

2.4.2) Complete the **decideMove()** method using Random().nextInt(4) to generate a random number between 0 and 3 (inclusive) and map this random number to one of the strings "up", "down", "left", or "right" to determine the monster's move to ensure that each direction has an equal probability of being chosen.

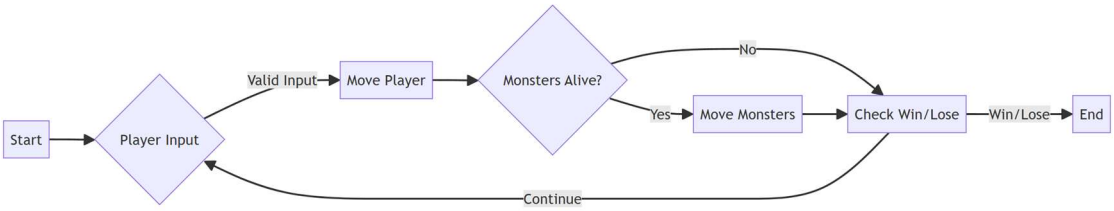
2.4.3) Implement the **Game** class and **GameLogic** class. I have finished

the following work:

- In the Game class, the **nextRound(String input)** method can handle user input by loop, move the player by using the **moveCharacter()** method, and move living monsters(health>0) by using the **decideMove()** method in the GameLogic class.
- Use **getMap()** method to print the map out every term the characters moved in the Game class.

2.4.4) Make a character unable to move to a place where stands another character. I employed the hasOccupiedCharacter() method to ensure it. If a monster attempts to move to a location occupied by another monster, the program prints "Monster already there so it cannot move," and the monster remains in its original position.

After completing the above work, Task 4 now enhance the GameLogic class to provide warning messages when characters attempt invalid moves. The decideMove() method now ensures equal probability for monster movements in different directions. The Game class manages player and monster movements, preventing characters from occupying the same space and printing informative messages.




## 2.5 Task 5: Implement the remaining game logic

2.5.1) Each round the **main()** method in class **RunGame** will call the **nextRound()** method in class Game. I added a for-loop in the nextRound() method to iterate each character and print their health condition to the console by the sayName() and getHealth() methods.

2.5.2) Since the attack between player and monster are via moving to the target's location, in the hasOccupiedCharacter() method, when detecting whether a monster attempts to move to a location occupied by another monster, I added a conditional statement. When the character itself is player and the target character is a monster, call the **huntCharacter()** method like

"character.hurtCharacter(otherCharacter);"

Therefore, the player can attack the monster by moving to their location.

2.5.3) By using the same approach in Task 5.2, when the character itself is a monster and the target character is a player, call the huntCharacter() method, then the monster can attack the player successfully.

2.5.4) Each time calling the hurtCharacter() method, the steps will be as follows:
Check whether the target character's health is equal or less than zero.
If so, if the character is monster, set its location in the map from "%" to **"x"**, indicating the monster's death.
The next time when player trying to move to this location, "Character already dead" will be printed to the console.

```
if (otherCharacter instanceof Monster && otherCharacter.getHealth() <= 0) {
    System.out.println("Character already dead");
    return true;
```

```
else if (otherCharacter instanceof Monster && character instanceof Player) {
    character.hurtCharacter(otherCharacter);
    if (otherCharacter.getHealth() <= 0) {
        gameMap.layout[otherCharacter.row][otherCharacter.column] = "x";
    }
    return true;
```

2.5.5) Each time the **nextRound()** method is called, I check whether the player win or lose.

- In the gameWin() method, I iterate every character whose index greater than 0 in the characters list. If there is a character that health is greater than zero, return false, otherwise return true.
- In the gameLoss() method, check whether the health of the first character in characters list is less than zero, if so, return true, otherwise return false.

In this way, each time the nextRound() method is called, and:

- if the return value of gameWin() is true, print "YOU HAVE WON!"
- if the return value of gameLoss() is true, print "YOU HAVE DIED!".
- Only when both of the methods return false, the nextRound() method will return true, which keep the game goes on.

```
Round 31

down
Player is moving down
Successful attack!

Health Player:40
Health Monster1:0
Health Monster2:50
Health Monster3:0

....xx
......
...*%.
......
```

```
Round 32

right
Successful attack!

Health Player:20
Health Monster1:0
Health Monster2:50
Health Monster3:0

....xx
......
...*%.
......
```

```
Round 33

right
Successful attack!
Character already dead

Health Player:20
Health Monster1:0
Health Monster2:0
Health Monster3:0

YOU HAVE WON!
....xx
......
...*x.
......
Game Over!
```

# 3. References

The code presented in this report is developed solely based on my own knowledge and understanding of the concepts. No external references, tutorials, or code snippets were consulted during the development of this code.