A black text on a white background

Description automatically generated

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SCHOOL OF ENGINEERING AND TECHNOLOGY**

**COURSEWORK FOR THE**

**BSC (HONS) INFORMATION TECHNOLOGY; YEAR 1**

**BSC (HONS) COMPUTER SCIENCE; YEAR 1**

**BSC (HONS) INFORMATION TECHNOLOGY (COMPUTER NETWORKING AND**

**SECURITY); YEAR 1**

**BSC (HONS) SOFTWARE ENGINEERING; YEAR 1**

**ACADEMIC SESSION 2023; SEMESTER 2,3,4**

**PRG1203: OBJECT ORIENTED PROGRAMMING FUNDAMENTALS**

**DEADLINE: 31 JULY 2023 11:59PM (Monday)**

**INSTRUCTIONS TO CANDIDATES**

* This assignment will contribute 20% to your final grade.
* This is a group (maximum 5 students) assignment

**IMPORTANT**

The University requires students to adhere to submission deadlines for any form of assessment. Penalties are applied in relation to unauthorized late submission of work.

Any work submitted after the deadline, or after any period of extension granted shall be marked as a Fail or awarded a zero.

**Academic Honesty Acknowledgement**

“I Lim Xiwei, Tham Rou Yi, Lim Xin Ping, Lam Hui Theng, Ng Jia Wen (student name). verify that this paper contains entirely my own work. I have not consulted with any outside person or materials other than what was specified (an interviewee, for example) in the assignment or the syllabus requirements. Further, I have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. I realize the penalties *(refer student handbook undergraduate programme)* for any kind of copying or collaboration on any assignment.”

A black background with a black square

Description automatically generated..................09/07/2023, A signature on a white background

Description automatically generated with medium confidence09/07/2023, **A close-up of a signature

Description automatically generated** 09/07/2023,A black background with a black square

Description automatically generated09/07/2023, 09/07/2023

(Student’s signature / Date)

**Group Number: 7**

**Team Members:**

|  |  |  |  |
| --- | --- | --- | --- |
| **No** | **Name** | **Student ID** | **Contribution %** |
| **1** | **Lim Xiwei** | **21045596** |  |
| **2** | **Tham Rou Yi** | **21024161** |  |
| **3** | **Lim Xin Ping** | **21060603** |  |
| **4** | **Lam Hui Theng** | **21029137** |  |
| **5** | **Ng Jia Wen** | **21057146** |  |

# Marking Scheme

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Criteria** | **Reference Marks** | | **Marks** | **Remarks** |
| Design (10%)  Implement good object-oriented design in solving the problem, with high modularity, maintainability and reusability. Able to identify appropriate classes and their relationships, complete the classes with appropriate attributes and methods. The design is well presented in UML class and class relationship diagrams. | **10** | **Excellent** |  |  |
| **7-9** | **Good** |
| **4-6** | **Average** |
| **1-3** | **Poor** |
| Coding (5%)  Fulfil all the functionalities and align to the design you have presented in the UML diagrams. Follow the best programming practices, such as naming convention, indenting, code structure, optimisation, with appropriate exception handling. Good user-friendliness. | **5** | **Excellent** |  |  |
| **4** | **Good** |
| **2** | **Average** |
| **1** | **Poor** |
| Additional Functionality (5%)  Add at least one additional enhancement or functionality to your program. Explain the rationale and reasoning by providing justification that supports the decision. | **5** | **Excellent** |  |  |
| **4** | **Good** |
| **2** | **Average** |
| **1** | **Poor** |
| **TOTAL** | **20** | |  |  |

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **No** | **Title** | **Page** |
| 1.0 | Introduction | 5 |
| 2.0 | UML Diagram | 6 – 8 |
| 3.0 | Classes and Methods | 9 – 14 |
| 4.0 | Additional Functionality | 15 |
| 5.0 | Test Cases | 16 – 19 |
| 6.0 | Boat Racing Game Program | 20 – 23 |

1. **INTRODUCTION**

This report presents an in-depth analysis and development of the "Boat Racing Game," a captivating and competitive Java program that promises an exhilarating racing adventure. This Java-based game showcases the application of object-oriented design, optimal maintainability, and reusability, making it an exemplary representation of software development best practices.

**OVERVIEW**

The Boat Racing Game is a two-player gaming application that emulates a thrilling boat race down a treacherous river. The winner for each game is determined based on which player's boat reaches the end of the river first. The game play is presented as shown below:

1. Both players are required to enter their name before the game starts.
2. Each player is assigned a boat, and their objective is to navigate through a 100-column river track, facing numerous challenges in the form of randomly scattered special elements which are Current (C), Trap (#), Frozen (@) and Multiplier ($).
3. Once their names are entered, the first player which is player A will roll the dice by pressing the “Enter” key. The number of dice rolled determines the number of steps of the player in each round. After player A has done with their turn, Player B will continue the game by rolling the dice next.
4. The movement of both players in each round is presented in the track. The track also displays the placement of the elements in every game. Once the player hits the element, their movement will be affected. Each functionality of the challenges and benefits are tabulated as shown below:

|  |  |
| --- | --- |
| **ELEMENTS** | **FUNCTIONALITY** |
| No Effects | |
| EMPTY ( \_ ) | At ‘\_’, player will advance forward according to the dice roll value. |
| Challenges | |
| TRAP (#) | When player encounters ‘#’, it is affected negatively, forcing the boat to move backward according to the randomly generated dice value. |
| FROZEN (@) | The player will skip their turn in the next round once they hit ‘@’. |
| Benefits | |
| CURRENT (C) | When a boat encounters ‘C’, it is influenced positively, causing the boat to move forward according to the randomly generated dice value. |
| MULTIPLIER ($) | The player’s movement will be multiplied by 2 based on the number of the dice rolled for their next turn. |

1. The game will continue until a winner is determined. The winner’s score will be recorded and presented in the game.
2. **UML DIAGRAM**

**UML CLASS RELATIONSHIP**

The UML class relationship has described how each class in the game program is related to each other and how they interact.

A diagram of a game

Description automatically generated

A summary of the relationship between the classes is listed below:

1. The “**Game**” class acts as the main controller in the program. The class mentioned has an aggregation relationship with the “Player” class, “TopScores” class and “River” class. To specify, the game has a river which acts as the track of the game, and it will have 2 players to start the gameplay. The game will have a list of top 5 players’ record in the top score list.
2. The “**TopScores**” class is associated with the “Player” class object, representing the player who achieved the score. This indicates that the relationship between “TopScores” class and the “Player” class is aggregation as the Top Score list has a record of 5 players’ info.
3. The “**River**” class has an aggregation relationship and a one-to-many relationship with the “Element” class, whereby the” River” class contains one or more “Element” objects. To illustrate, the river track will have multiple elements located in it during the game.
4. The “**Element**” class is a parent class of the “**Frozen**”, “**Trap**”, “**Empty**”, “**Current**” and “**Multiplier**” child classes. This relationship between the parent class and child class is defined as inheritance relationship, where the child classes inherit the attributes and methods of the “Element class” and add their own unique attributes and methods specific to their type of element. There also exist a one-to-many relationship between the parent and child classes.
5. The “**Dice**” class does not have a relationship with any classes as the class only consists of static object and static method. The static method will only be called in any class once the dice functionality is required in that particular class.

**UML CLASS**

A screenshot of a computer program

Description automatically generated A screenshot of a computer program

Description automatically generated

**Figure 1. Class Diagram of Game & TopScores**

A screenshot of a computer program

Description automatically generated A screenshot of a computer program

Description automatically generated

**Figure 2. Class Diagram of Player**

A screenshot of a computer program

Description automatically generated A screenshot of a computer

Description automatically generated

**Figure 3. Class Diagram of River & Element**

A screenshot of a computer program

Description automatically generated

**Figure 4. Class Diagram of Frozen, Trap, Empty, Current, Multiplier**

A screenshot of a computer code

Description automatically generated

**Figure 5. Class Diagram of static class Dice**

1. **CLASSES & METHODS**

**PLAYER CLASS**

The "Player" class manages each player's name, position on the game board, number of turns, and other gaming data. It provides a number of constructors for various player creation scenarios, such as a parameterized constructor to establish the player's name and turn count and a constructor for game setup that asks the player for their name and provides a character symbol. Access to player characteristics is made possible using getter and setter methods. The "toString()" functions to return the data of the winning player at the end of the game.

|  |  |
| --- | --- |
| **Method** | **Explanation** |
| info(char sym) | Get the player's name:   1. Prompt user to enter name when game starts 2. Empty entries are not allowed |
| play() | Simulates player’s turn. |
| currentEffect() | To trigger an extra dice roll (auto-generated) when player hit a current, record and print out new position. It will not go over the track (max position = index[99]). |
| trapEffect() | When a player landed on a trap symbol, the player moves backward based on the randomly generated dice roll, but will not go out of the track (min position = index[0]). |
| **Additional Functionality** | |
| isFreezeNextTurn() | Returns a boolean value indicating whether the player's next turn is to be skipped due to the freezing effect. |
| setFreezeNextTurn(boolean freezeNextTurn) | Sets the player's frozen condition. |
| isMultiplyNextTurn() | Provides a boolean result indicating whether the player's next turn will be multiplied because of the effect. |
| setMultiplyNextTurn (boolean multiplyNextTurn) | Sets the player's multiply status. |
| isResetFrozen() | Determines if the '@' element's status needs to be reset once the player leaves the location by returning a boolean value. |
| setResetFrozen(boolean resetFrozen) | '@' element status is set, indicating if the element needs to be reset. |

**TOPSCORE CLASS**

The top players in the game are managed by the "TopScores" class. It uses a String variable named "fileName" to hold the name of the file where the top scores will be recorded, and an ArrayList called "topPlayers" to store Player objects that represent the top players. Overall, the "TopScores" class makes sure that top player scores are efficiently managed and tracked, improving the game's excitement and competition.

|  |  |
| --- | --- |
| **Method** | **Explanation** |
| updateScores(Player winner) | Ensures that the "topPlayers" list only contains the top 5 scores and adds a new Player object (the winner) to it every round. The lowest score is removed using sorting and comparisons if the list size is more than 5. |
| getFormattedTopScores() | The top players' names and scores are produced in a prepared string. |
| displayScores() | Function reads the top scores from the given file and displays them. |
| readScoresFromFile() | Gets the top scores from the file. |
| writeScoresToFile() | Updates the top scores after each game by writing the formatted top scores to the given file. |

**RIVER CLASS**

The "River" class controls the river's elements and serves as the river's representation in the game. It helps to manage the game's dynamic and ever-changing environment by placing pieces in optimal locations for players to go through the river and interact with. The class has a Random object named "r" that creates random numbers and an ArrayList called "eList" that keeps each element of the river.

|  |  |
| --- | --- |
| **Method** | **Explanation** |
| River() | Constructor that fills 100 "Empty" entries into the eList ArrayList. Randomly creates 40 random components, such as "Current," "Trap," "Multiplier," and "Frozen," then distributes them all through the game board in various combinations. |
| River(ArrayList eList) | A method with parameters that sets the eList ArrayList's items according to a given list. |
| geteList() | The eList ArrayList holding the river's elements may be obtained using the getter function. |
| seteList(ArrayList <Element> eList) | A new list of elements can be added using the setter method for the eList ArrayList. |
| setRiverElement(int position, Element element) | An element may be entered into a certain place in the river. |
| printRiver() | Loops through an "eList" and prints the symbols of each element one at a time to show the present condition of the river. |

**DICE CLASS**

The “Dice” utility class represents and simulates the rolling of a normal six-sided dice. It has a static “Random” object named “random” to generate random numbers and a static integer variable named “dice” to hold the dice roll result.

|  |  |
| --- | --- |
| **Method** | **Explanation** |
| rollDice() | It simulates the roll of a six-sided dice by using “random.nextInt(6) + 1” to obtain a random number between 1 and 6 (inclusive). The rolled value is saved in the “dice” variable, and the method returns the rolled value. |
| getDICE() | Getter method that simply returns the value stored in the “dice” variable at the time. It enables other parts of the programme to obtain the outcome of the most recent dice roll. |

**ELEMENT CLASS**

A fundamental component of a game board is represented by the "Element" class, which has the properties "position" and "symbol." It has two constructors that are default constructor, which has no parameters, and the parameterized constructor, which lets you provide the position and symbol when producing "Element" objects. The "position" and "symbol" characteristics may be accessed and changed using the class's getter and setter methods.

|  |  |
| --- | --- |
| **Method** | **Explanation** |
| Element() | Default constructor of the class. |
| Element(int position, char symbol) | Constructor with parameters for setting the element's location and symbol. |
| setPosition(int position) | A new location for the element. |
| getPosition() | Gives the element's current position value. |
| setSymbol(char symbol) | To assign a new symbol to the element. |
| getSymbol() | Gathers the element's current symbol. |

In the end, the "Element" class functions as the foundation for the game's other unique element classes, including “Empty”, "Current", "Trap", "Multiplier", and "Frozen". In order to organise and control the many elements on the game board, it offers a standard framework for all elements to have a place and a symbol.

**EMPTY CLASS**

In the game board, space with no special effects is represented by the "Empty" class. The class offers an empty foundation on which other specialised features can be added during the development of the river, adding to the overall variety and complexity of the game.

|  |  |
| --- | --- |
| **Method** | **Explanation** |
| Empty() | The "setSymbol('\_')" method is used to construct an instance of a blank space with the character  underscore. |
| Empty(int position, char symbol) | It uses the syntax "super(position, symbol)" to send the location and symbol to the superclass "Element" constructor. |

**CURRENT CLASS**

The "Current" class, represents the moving waves that facilitates players’ movement, is an essential part of the game. With its two constructors, it provides the ability to produce current pieces randomly or purposefully placed. This element act as a power-up, and serves to enhance players’ excitement throughout the game.

|  |  |
| --- | --- |
| **Method** | **Explanation** |
| Current() | It creates a random position for the river's current element. It then uses the "setPosition()" function to set this location to a random value between 0 and 98 that was obtained using a "Random" object. Finally, it uses the "setSymbol()" function to set the symbol "C" for the currently selected element. |
| Current(int position, char symbol) | It uses "super(position, symbol)" to send the inputs position and symbol to the constructor of the superclass "Element". |

**TRAP CLASS**

In the game, a trap element ‘#’ is represented by the "Trap" class, which is a subclass of the "Element" class. To build trap instances, it has two constructors, one whereby the position is randomly set another is manually entered. The trap element is a distinctive obstacle in the game and it increases gameplay complexity and difficulty in winning the game.

|  |  |
| --- | --- |
| **Method** | **Explanation** |
| Trap() | Using a Random object and "r.nextInt(98 + 1) ", the constructor selects a random position for the trap element in the river between 1 to 99. Index 0 is not included as a trap at the first position will create an infinite loop and causes an error.  The "setPosition()" function is used to assign the location, while the "setSymbol('#') " method is used to symbolise the trap element. |
| Trap(int position, char symbol) | The constructor of the superclass 'Element' receives the inputs 'position' and ‘symbol', which are passed using the syntax' super(position, symbol)'. With this constructor, it is possible to insert unique instances of the "Trap" element that according to its position and symbol. |

**GAME CLASS**

The centre of the game simulation is the "Game" class. The management of the game's flow, player turns, and interactions with the river elements are all within the class. The class has a few methods that handle various gaming functions, including moving players, examining the results of landing on certain symbols, updating the river, and controlling player turns. It is an important class as it controls the game mechanics.

|  |  |
| --- | --- |
| **Method** | **Explanation** |
| Game() | Constructor for the "Game" class, initializes the "River," "currentPlayer," "opponent," and "topScores" objects for gameplay. |
| setPlayerSymbol (Player player) | Sets the player's symbol on the river grid to display their present location. |
| setOriSymbol (Player player, char symbol) | The specified symbol is replaced with the original symbol on the river grid at the player's current location. Used after to reset the grid before the player’s next dice roll/ next position. |
| getCurrentSymbol (Player player) | Returns the symbol on the river grid at the player's current position. |
| getIndexSymbol (Player player) | Returns the component object that is located on the river grid of the player's current position. |
| start() | Starts the game and continues until one player reaches position 100 in the main game loop. |
| playerTurn() | Controls all aspects of the game's turn-taking process, including rolling the dice, moving the player, and assessing the impact of a player's landing on various symbols. |
| playerMove() | Handles player movement. |
| checkEffect() | Based on the symbol the player lands on, determines the relevant actions, such as currents, traps, frozen areas, and multipliers. |
| playerProcedure() | Before going on to the following turn, prints the player's location and update the river grid appropriately. |

**TESTGAME CLASS**

The "TestGame" class functions as the game's starting point. An instance of the "Game" class is created inside the "main" function and given the name "game." Upon instantiation, the constructor of the "Game" class is called, which sets up the players, creates the river, and starts the top scores for the game.

|  |  |
| --- | --- |
| **Method** | **Explanation** |
| public static void main(String[] args) | When the programme is run, the main method is executed. |

1. **ADDITIONAL FUNCTIONALITY**

**FROZEN CLASS**

Frozen is one of the additional features in our game. It functions to “freeze” the player on its landed position, so that they cannot move for their next turn. In other words, if a player encounters the “Frozen” element, their next turn will be skipped. Hence, it will increase the difficulty in the overall gameplay and level design, making the game even interesting.

Same as the other elements, the "Frozen" class is a subclass of the "Element" class that represents frozen locations on the game board. The class has two constructors.

|  |  |
| --- | --- |
| **Method** | **Explanation** |
| Frozen() | Default constructor. When the game first run, the method generates a random location on the board for the frozen element by utilising a "Random" object, "r," and "r.nextInt(99)" to obtain a random number between 0 and 98. This location is assigned by the "setPosition()" function, and the symbol of the element is set as "@". |
| Frozen(int position, char symbol) | Parameterized constructor that allows instances of the "Frozen" element to be allocated specific positions and symbols as needed. This constructor is helpful when specified frozen places are required for level design or specialised game dynamics. |

**MULTIPLIER CLASS**

Multiplier is our second additional functionality in the game. The “Multiplier” class adds a fun gaming feature by giving players the chance to multiply their progression. When a player landed on the Multiplier ‘$’, their next dice roll will be multiplied by 2, which allows them to advance further and faster to the winning point. Not only does it balance out the challenges in the game, but it also builds anticipation and excitement for the players to experience its effects.

Similarly, the “Multiplier” is one of the child class of the “Element” class, and also consists two constructors.

|  |  |
| --- | --- |
| **Method** | **Explanation** |
| Multiplier() | The "Multiplier" element in the river is generated at random by the default constructor using a "Random" object and "r.nextInt(99)" to get a random value between 0 and 98. The symbol "$" is then set using "setSymbol('$')" and the position is then assigned using the "setPosition()" function. |
| Multiplier(int position, char symbol) | The "Multiplier" element may be created with specified instances that have preset locations and symbols due to the parameterized constructor. "super(position, symbol)" is used to send the supplied location and symbol to the constructor of the superclass "Element". |

1. **TEST CASES**

|  |
| --- |
| **Case 1:** **Player Movement without Special Effects** |
| **Case 2:** **Player Movement with Current Effect** |
| **Case 3: Player Movement with Trap Effect** |
| **Case 4: Player Movement with Frozen Effect** |
| **Case 5: Player Movement with Multiplier Effect** |
| **Case 6: Position becomes Empty after Player Move Away** |
| **Case 7: ‘P’ Symbol when Both Players are at Same Position**  . |
| **Case 8: Detect Effect at New Position After Previous Effect** |
| **Case 9: Element Symbol is Restored After Player Move Away** |
| **Case 10: Player Reach the End of the River** |
| **Case 11: Displaying Top Player Scores** |
| **Case 12: Displaying Updated Top Player Scores**      **Figure 1: Program Console Leaderboard**    **Figure 2: Text File Leaderboard** |

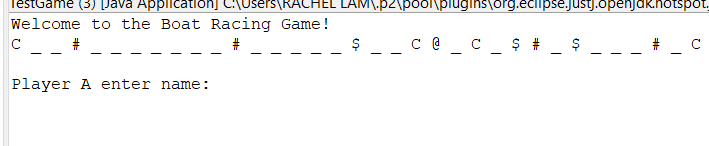
1. **BOAT RACING GAME PROGRAM**

The "Boat Racing Game" is a Java program that is designed and demonstrated with object-oriented principles and best practices in software development. It offers an exhilarating gaming experience with its competitive gameplay and dynamic river environment. The game allows two players to participate in a thrilling boat race down a treacherous river, facing challenges and benefits along the way.

The game's classes and their relationships are designed in a way to promote reusability and maintainability. The key classes in the game are well-defined and serve specific purposes. The "Player" class manages player data and turn-taking, the "TopScores" class tracks and updates the top players, the "River" class represents the river track with various elements while the "Element" class serves as the parent class for different river elements like traps, currents, and multipliers.

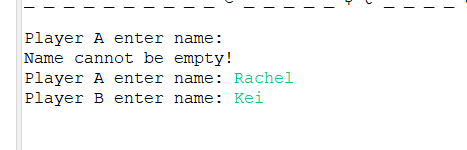
The game also features challenges and benefits functionality such “Current” elements which allow the player to move forward with a certain number of steps; “Trap” elements which will generate a random dice value to alter the player’ move to move backwards; "Frozen" elements, which skip the player's next turn; and "Multiplier" elements that allow players to double up their movement. The "Dice" class, while not directly related to other classes, serves the essential function of simulating dice rolling, contributing to the randomness of the game.

With the combination of different classes in the game, the output of the gameplay is presented as below:



*Figure (1): The start of the game*

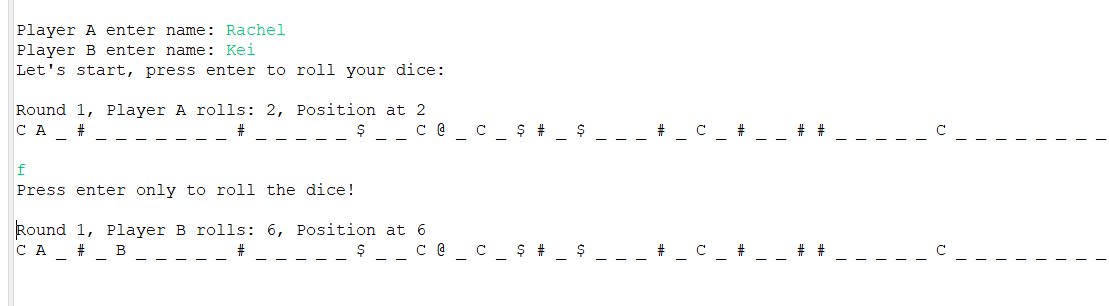
1. In Figure (1), the program will start off with the display of statement “Welcome to the Boat Racing Game!” and the river track with the 100 elements. The elements including “Current” (C), “Trap” (#), “Multiplier” ($), “Empty” (\_) and “Frozen” (@) would be placed randomly in the river track.





*Figure (2): Players enter their names*

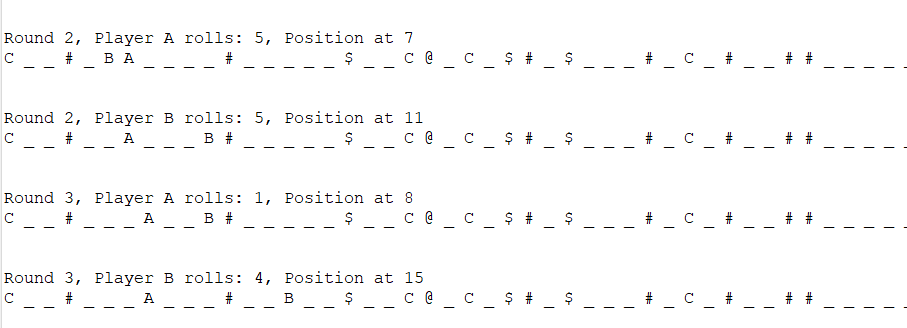
1. The program would prompt the players to enter their desirable name as shown in Figure (2) above. Player A which is determined as the first player would be required to type their name into the program and the following step would be repeated with Player B which is the second player. If both or one of the players did not enter their name, the program would execute the statement “Name cannot be empty!” in order to alert the player to key in their name.





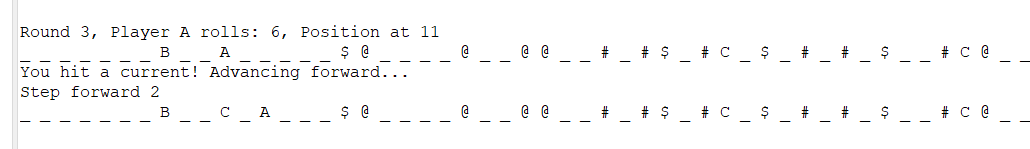
*Figure (3): The start of the gameplay*

1. In Figure (3), once both players have entered their names, the game officially begin. The program would prompt Player A to hit the “Enter” key to roll the dice. If the player hit any key other than the “Enter” key, it would alert the player to follow the instruction.



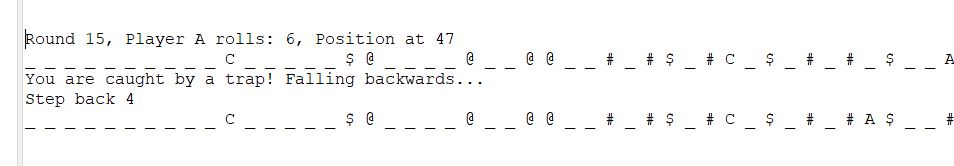
*Figure (4): Each round of the gameplay*

1. In Figure (4), once Player A has hit the “Enter” key, the program would generate a random number from 1 to 6. The number will determine the player’s movement and the current position of Player A would be displayed. Each round would end with the two players completed the dice rolling.





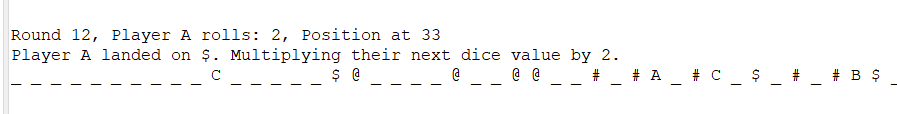
*Figure (5): Player hits a Current*





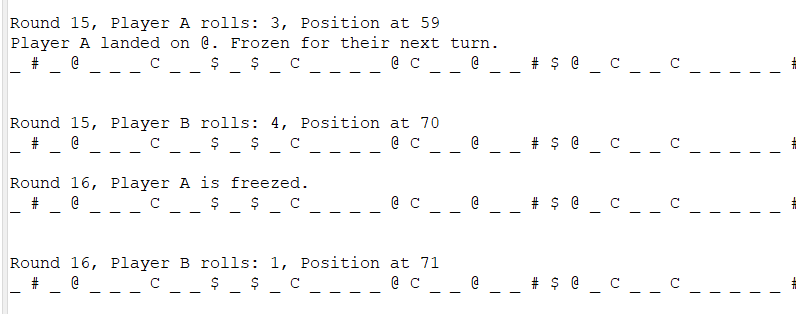
*Figure (6): Player hits a Trap*

1. In Figure (5), whenever a player landed on a “Current”, the program would inform the player that they have hit the “Current” benefits and their movement would be added forward based on the randomly generated number of steps. Vice versa, if the player has landed on “Trap”, the player’s movement would step backwards depending on the randomly generated steps as it shown in Figure (6).



*Figure (7): Player hits a Multiplier*

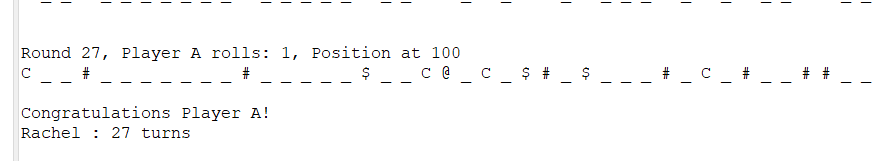
1. In Figure (7), during the gameplay, if the player landed on the “Multiplier”, the program would inform the player with the number of dice rolled and their current position. Then, player’s position will be automatically multiplied by the number of the dice rolled.





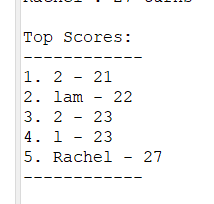
*Figure (8): Player hits a Frozen*

1. In Figure (8), as the player’s movement ends up with “Frozen”, during the next round, the player would lose their chance to roll the dice and their position would remain the same as the previous position.



*Figure (9): The winner of the game*

1. In Figure (9), the gameplay would officially end once a player has reached position 100. The round taken for the winner to reach the goal would be recorded.



*Figure (10): Top score of the winners*

1. In Figure (10), the program would end with the display of the top score. The top score would display the number of rounds for the previous winner to get to position 100. The ranking would be based on the player who use least number of rounds to get to the goal.