Functionality -

Our program successfully runs and compiles without errors. Keyboard actions that have not been defined in our code do not affect the runtime of the program. All potential user inputs (3D or 2D maze, dimensions, key, enemies or no enemies) have robust statements checking to ensure that all inputs are within bounds or are the correct type. If incorrect, an error message will appear to inform the user that they need to specify another value within bounds. If the user chooses to not input or change one or more aspects of the maze, there are default values that will generate the maze. If a PDF or other type of file is submitted for the maze image instead of a JPEG, the program will still run with no problems in functionality.

Design -

In our program, we have two different renderings of a maze: 2D and 3D. In our 2D graphics class, the code is succinct and produces an interactive maze. For our 3D graphics class, because there are so many different walls and the enemies/health graphics require many parameters, the code is less compact. We also included code for different shading and visuals that give the maze a more genuine “3D” effect. For the actual maze generation, we implemented class BacktrackerDS. This created the maze by moving through the cells to ensure each one was “visited”, and if not, it backtracks to create a path and ensure that every cell could be visited as well as guaranteeing that there is one unique solution.

Creativity -

While there are plenty of random maze generators online, our 3d visuals and monsters and health that you fight along the way make our project a little more unique. One aspect that is certainly very interesting is how we dealt with not using something that was meant for 3d graphics. Instead of using a java 3d graphics library, we used our knowledge of the 2d graphics, and some tricking eye illusion business to make it appear 3d. There are quite a few 3d mazes that I found online, many of which had similar concepts, however, I think the controls for our maze and our graphics were much more pleasing. Also, again, our maze can have monsters that you fight, which is cool.

Sophistication -

We drew from many of the labs and the skills we learned from those labs for this project including keylisteners, graphics, linked list data structures, input and output to a text file, and pseudo-random number generators. A few things that we did not learn in class or in labs that we used for this project were JButtons as well as JTables. Another sophisticated part of our project is the actual maze generator. It uses depth first search combined with a data structure that allows for backtracking to create a randomly generated maze.

Broadness -

For broadness, we used 1, 4, 5, 6, and 7.

1. Java libraries we haven't seen in class

In the leaderboard class, we import and use the DefaultTableModel class to modify our JTable so it can't be edited. Also, in that class, we use GridBagConstraints to split the frame into multiple panels that each take up a weighted section of the frame. In the FrontPageGraphics class, we use a BufferedImage to set the background of the main menu to an external jpeg file.

4. User-defined data structures

We created a BacktrackerDS class that helps in the maze generation process by keeping track of which cells the maze generator has visited. It serves as a modified linked list that stores node objects (representing cells) with a boolean if they were visited, an x-coordinate, a y-coordinate, and the following node.

5. Built in data structures

In the leaderboard class, we use an ArrayList to represent the leaderboard to let the class add a new player to the leaderboard easily. By reading the text file where the leaderboard is stored and translating it into an arraylist, the class is able to work with it much more easily.

6. File input/output

In the leaderboard class, we store the leaderboard in external text files outside of the program so all scores can be logged and kept. When a player finishes the 3D maze, the leaderboard class will read the players' data from the leaderboard text file and display it in a GUI. If the player then decides to submit their score to the leaderboard, the class will write their name and time into the text file.

7. Randomization

We use randomization in the maze class where we generate a random maze based on the seed that the player inputs. The maze generator starts in a random spot on the maze and randomly chooses which direction to move in. When the maze generator chooses which direction to move in, it will cut the wall behind it. And if the generator gets into a dead end, it will backtrack until it has a new direction to go in. After visiting every square once, the maze will be completed and it will have ensured a solution because every square is now able to visit any other square in the maze.

Code Quality -

For code quality, we did a good job of commenting through our code using javadoc formatting for all the classes, member variables, constructors, and methods with single line comments throughout explaining certain portions of the code. I also think our code organization across files was done well. We subdivided the program into tasks to be handled by different classes which made intuitive sense and allowed us to better focus on different sections of the code. I think our code is decently clean in terms of how code is organized into various methods inside the class. Although, for some of the more complex classes such as the 3D graphics class, the methods could have been named more descriptively and it might be confusing to someone who's looking at the code for the first time.

I think our final product switches from one frame to the next well with the main menu, the game window, and the leaderboard, although the graphics could have been nicer to make it look more professional.