Documentation of Maze Game

I. How the game works

The Maze Game is a Java-based application that challenges players to navigate through a randomly generated maze to reach the exit. The game offers two primary modes: **Single Player** and **Competitive Mode**. It incorporates features such as maze generation, AI solving, maze regeneration, saving/loading game states, and visual rendering using Turtle.

Game Modes

1. **Single Player Mode**:

- The player navigates through the maze to reach the exit.

- The player can request hints (next 10 steps) or the full solution.

- The game tracks the player's moves and elapsed time.

2. **Competitive Mode**:

- The player competes against an AI to reach the exit first.

- Players take turns making moves, and the game tracks moves for both the player and the AI.

- The AI follows a pre-calculated optimal path to the exit.

Features

- **Maze Generation**: The maze is generated using a Depth-First Search (DFS) algorithm, ensuring a unique and solvable maze every time.
- **AI Solver**: The AI uses a DFS-based algorithm to find the optimal path from the entrance to the exit.
- **Maze Regeneration**: The maze regenerates every 60 seconds, preserving the player's position if possible.

- **Save/Load Game**: Players can save their progress and load it later to continue the game.
- **Visual Rendering**: The maze is drawn using a Turtle graphics library, with colors for walls, players, entrance, exit, and solution paths.
- **Timer and Move Counter**: The game displays elapsed time and the number of moves made by the player (and AI in competitive mode).

Winning Conditions

- **Single Player Mode**: The player wins by reaching the exit. The game displays the elapsed time and number of moves.
- **Competitive Mode**: The player or AI wins by reaching the exit first. The game displays the number of moves for both players and the winner.

II. Code Overview

1. Class Overview

Maze Class

The `Maze` class is the central component of the Maze Game, responsible for managing maze generation, player movements, AI interactions, game modes, saving/loading game states, and rendering. It leverages other supporting classes such as `Cell`, `Player`, and `Point` to encapsulate various aspects of the game.

Key Responsibilities:

- **Maze Generation**: Creates a unique, solvable maze using the Depth-First Search (DFS) algorithm.
- **AI Solving**: Implements an AI that can solve the maze optimally.
- **Game Management**: Handles game modes, player turns, and game state transitions.
- **Rendering**: Uses a Turtle graphics library to visually represent the maze, players, AI, and solution paths.
- **Persistence**: Allows saving and loading game states for continued play.

---

2. Constants

The `Maze` class defines several constants to configure game settings and rendering properties:

- **Maze Dimensions:**

- `MIN_SIZE`: Minimum allowed size for maze dimensions.

- `MIN_CELL_SIZE`: Minimum pixel size for each cell in the maze.

- **Colors (RGB):**

- `PLAYER_COLOR`: Color representing the human player (Blue).

- `AI_COLOR`: Color representing the AI player (Red).

- `WALL_COLOR`: Color of the maze walls (Black).

- `ENTRANCE_COLOR`: Color of the maze entrance (Green).

- `EXIT_COLOR`: Color of the maze exit (Red).

- `TIMER_TEXT_COLOR`: Color of the timer and move counters (Black).

- `PATH_COLOR`: Color used to display solution paths (Green).

- **Timers:**

- `REGENERATION_TIME`: Interval (in seconds) at which the maze regenerates automatically.

```
    private final int MIN_SIZE = 1;

    private final int MIN_CELL_SIZE = 10;



    private final int PLAYER_COLOR = 0x0000FF;     // Blue

    private final int AI_COLOR = 0xFF0000;         // Red

    private final int WALL_COLOR = 0x000000;       // Black

    private final int ENTRANCE_COLOR = 0x00FF00;   // Green
```

```
    private final int EXIT_COLOR = 0xFF0000;      // Red

    private final int TIMER_TEXT_COLOR = 0x000000;// Black


    private final int REGENERATION_TIME = 60; // seconds


    private final int PATH_COLOR = 0x00FF00; // Green path for solv
```

---

3. Fields

The `Maze` class maintains the following fields to manage the game state:

- **Maze Dimensions:**

- `width`: Number of cells horizontally.

- `height`: Number of cells vertically.

- `cellSize`: Pixel size of each cell for rendering.

- **Maze Data:**

- `grid`: 2D array of `Cell` objects representing the maze structure.

- `random`: Instance of `Random` for generating random numbers during maze creation.

- **Entrance and Exit:**

- `entrance`: `Cell` object representing the maze's entrance.

- `exit`: `Cell` object representing the maze's exit.

- **Players:**

- `player`: Instance of `Player` representing the human player.

- `aiPlayer`: Instance of `Player` representing the AI player in competitive mode.

- **AI Solution:**

- `aiSolution`: List of `Character` representing the full solution path calculated by the AI.

- `aiPlannedMoves`: List of `Character` representing the AI's planned moves.

- **Rendering:**

- `turtle`: Instance of `Turtle` graphics library used for drawing the maze and players.

- **Timers:**

- `gameTimer`: `Timer` object tracking the elapsed game time.

- `displayTimer`: `Timer` object potentially used for periodic updates (not detailed in code).

- `regenerationTimer`: `Timer` object triggering maze regeneration at set intervals.

- **Game State:**

- `elapsedTime`: Total time elapsed since the game started, in seconds.

- `isGameRunning`: Boolean flag indicating if the game is currently active.

- `isRegenerating`: Boolean flag indicating if a maze regeneration is in progress.

- **Competitive Mode:**

- `isCompetitiveMode`: Boolean flag indicating if the game is in competitive mode.

- `humanMoveCount`: Counter for the number of moves made by the human player.

- `aiMoveCount`: Counter for the number of moves made by the AI.

- `isHumanTurn`: Boolean flag indicating if it's the human player's turn in competitive mode.

- **Input Handling:**

- `scanner`: `Scanner` instance for reading user input from the console.

```
    // Maze dimensions
    private int width;
    private int height;
```

```
    private int cellSize;



    // Maze data

    private Cell[][] grid;

    private final Random random;



    // Entrance and exit cells

    private Cell entrance;

    private Cell exit;



    // Player and AI

    private Player player;

    private Player aiPlayer;



    // AI solution path

    private final List<Character> aiSolution;

    private List<Character> aiPlannedMoves;



    // Turtle for drawing

    private transient Turtle turtle;



    // Game timers

    private transient Timer gameTimer;
```

```
    private  transient Timer displayTimer;
    private transient Timer regenerationTimer;


    // Game state
    private int elapsedTime;
    private boolean isGameRunning;
    private boolean isRegenerating;


    // Competitive mode
    private boolean isCompetitiveMode;
    private int humanMoveCount;
    private int aiMoveCount;
    private boolean isHumanTurn;


    // Scanner
    private transient Scanner scanner = new Scanner(System.in);
```

---

4. Constructor

`Maze(Turtle turtle, int width, int height, int cellSize)`

- **Purpose**: Initializes a new `Maze` instance with specified dimensions and rendering settings.
- **Parameters**:

- `turtle`: The `Turtle` instance used for rendering the maze and players.

- `width`: The width of the maze in cells.

- `height`: The height of the maze in cells.

- `cellSize`: The pixel size of each cell.

- **Process**:

1. Assigns and validates maze dimensions, ensuring they meet minimum size requirements.

2. Initializes the maze grid by calling `initializeGrid()`.

3. Initializes AI solution lists (`aiSolution` and `aiPlannedMoves`).

4. Sets competitive mode flags and move counters to default values.

```
    public Maze(Turtle turtle, int width, int height, int cellSize
        this.turtle = turtle;
        this.width = Math.max(MIN_SIZE, width);
        this.height = Math.max(MIN_SIZE, height);
        this.cellSize = Math.max(MIN_CELL_SIZE, cellSize);
        this.random = new Random();


        this.grid = initializeGrid();
        this.aiSolution = new ArrayList<>();
        this.aiPlannedMoves = new ArrayList<>();


        this.isCompetitiveMode = false;
        this.humanMoveCount = 0;
        this.aiMoveCount = 0;
```

```
        this.isRegenerating = false;

    }
```

---

5. Primary Methods

Primary methods are the core functionalities that drive the main features of the game. They often call upon helper methods to perform specific tasks.

5.1. Maze Generation

`generate()`

```
    public void generate() {
        resetGrid();
        Cell startCell = grid[0][0];
        startCell.visited = true;

        List<Cell> stack = new ArrayList<>();
        stack.add(startCell);

        while (!stack.isEmpty()) {
            Cell current = stack.get(stack.size() - 1);
            Cell next = getUnvisitedNeighbor(current);

            if (next != null) {
                removeWalls(current, next);
```

```
                next.visited = true;

                stack.add(next);

            } else {

                stack.remove(stack.size() - 1);

            }

        }

        createEntranceAndExit();

    }
```

- **Purpose**: Generates a new maze layout using the Depth-First Search (DFS) algorithm.
- **Process**:

1. Calls `resetGrid()` to initialize the maze grid.

2. Starts DFS from the initial cell (`grid[0][0]`), marking it as visited.

3. Uses a stack (`List<Cell>`) to traverse cells, carving out paths by removing walls between the current and next cells.

4. Continues until all cells are visited.

5. Calls `createEntranceAndExit()` to define the maze's entrance and exit points.

5.2. AI Solving

`findSolution(Cell current, Set<Cell> visited, List<Character> currentPath)`

```
 private boolean findSolution(Cell current, Set<Cell> visited, List

        if (current == exit) {

            aiSolution.addAll(currentPath);

            return true;
```

```
        }

        visited.add(current);



        for (int i = 0; i < 4; i++) {

            if (!current.walls[i]) {

                int[] delta = getDeltaFromDirection(i);

                int newX = current.x + delta[0];

                int newY = current.y + delta[1];



                if (isValidCell(newX, newY) && !visited.contains(

                    currentPath.add(getCharFromDirection(i));

                    if (findSolution(grid[newY][newX], visited, cu

                        return true;

                    }

                    currentPath.remove(currentPath.size() - 1);

                }

            }

        }

        visited.remove(current);

        return false;

    }
```

- **Purpose**: Recursively searches for a path from the current cell to the exit using DFS.
- **Process**:

1. Marks the current cell as visited.

2. If the current cell is the exit, appends the current path to `aiSolution`.

3. Iterates through all possible directions (Up, Right, Down, Left).

4. For each direction without a wall, moves to the neighboring cell and continues the search.

5. Backtracks if no path is found in a direction.

`solveMaze()`

```
 private void solveMaze() {

         aiSolution.clear();

         Cell currentPos = getCell(player.x, player.y);

         findSolution(currentPos, new HashSet<>(), new ArrayList<>



         if (aiSolution.isEmpty()) {

             System.out.println("No solution found!");

             return;

         }

         System.out.println("Showing full path in green...");



         // Draw the path in green, from the player's current posit

         drawPathInGreen(currentPos, aiSolution);

     }
```

- **Purpose**: Initiates the maze-solving process and visually displays the full solution path.
- **Process**:

1. Clears any existing AI solutions in `aiSolution`.

2. Retrieves the player's current position (`Cell`).

3. Calls `findSolution()` to compute the solution path.

4. If a solution is found, calls `drawPathInGreen()` to visualize the path.

5. Notifies the player of the solution display.

`provideNext10Steps()`

```
 private void provideNext10Steps() {

         aiSolution.clear();

         Cell currentPos = getCell(player.x, player.y);

         findSolution(currentPos, new HashSet<>(), new ArrayList<>

         if (aiSolution.isEmpty()) {

             System.out.println("No solution available.");

             return;

         }

         int stepsToProvide = Math.min(10, aiSolution.size());

         List<Character> nextSteps = new ArrayList<>(aiSolution.sub

         System.out.println("Showing next " + stepsToProvide + " st

         drawPathInGreen(currentPos, nextSteps);

     }
```

- **Purpose**: Displays the next 10 steps of the AI's solution path from the player's current position.

- **Process**:

1. Clears any existing AI solutions in `aiSolution`.

2. Retrieves the player's current position (`Cell`).

3. Calls `findSolution()` to compute the solution path.

4. Extracts the next 10 moves from `aiSolution`.

5. Calls `drawPathInGreen()` to visualize these moves.

6. Notifies the player of the displayed steps.

5.3. Maze Regeneration

_(##### **`regenerateMaze()`**

```
 private synchronized void regenerateMaze() {
         if (isRegenerating || !isGameRunning) return;
         isRegenerating = true;


         // Store current positions
         Point currentPlayerPos = new Point(player.x, player.y);
         Point currentAIPos = (isCompetitiveMode && aiPlayer != nu
                              ? new Point(aiPlayer.x, aiPlayer.y)
                              : null;


         // Store old walls
         boolean[][][] oldWalls = new boolean[height][width][4];
         for (int y = 0; y < height; y++) {
             for (int x = 0; x < width; x++) {
```

```
            oldWalls[y][x] = Arrays.copyOf(grid[y][x].walls, 4
        }
    }

    // Generate new maze
    generate();

    // Ensure path exists for both players
    if (!ensurePathExists(currentPlayerPos)
            || (isCompetitiveMode && !ensurePathExists(current
        // If no valid path, restore old maze
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                System.arraycopy(oldWalls[y][x], 0, grid[y][x]
            }
        }
        generate(); // Optionally try again
    }
    // Recalculate AI solution if competitive
    if (isCompetitiveMode && currentAIPos != null) {
        aiSolution.clear();
        aiPlannedMoves.clear();
        findSolution(grid[currentAIPos.y][currentAIPos.x], new
```

```
            aiPlannedMoves = new ArrayList<>(aiSolution);
        }

        // Update player positions
        player.x = currentPlayerPos.x;
        player.y = currentPlayerPos.y;
        if (isCompetitiveMode && aiPlayer != null && currentAIPos
            aiPlayer.x = currentAIPos.x;
            aiPlayer.y = currentAIPos.y;
        }

        // Redraw
        redrawMazeAndPlayers();
        isRegenerating = false;
        System.out.println("Maze regenerated! Keep going!");
    }
```

- **Purpose**: Regenerates the maze at set intervals while attempting to preserve player positions.
- **Process**:

1. Checks if regeneration is already in progress or if the game is running.
2. Saves current player (and AI, if in competitive mode) positions using `Point` instances.
3. Stores the current walls configuration to allow restoration if needed.
4. Calls `generate()` to create a new maze.

5. Ensures that paths exist from the saved positions to the exit using `ensurePathExists(Point start)`.

6. If no valid path exists, restores the old maze configuration.

7. Repositions players to their saved locations.

8. Redraws the maze and players.

9. Resets the regeneration flag and notifies the player.

5.4. Save and Load Game

`saveGame()`

```
public void saveGame() {
        try (ObjectOutputStream out = new ObjectOutputStream(new I
            out.writeInt(this.width);
            out.writeInt(this.height);
            out.writeInt(this.cellSize);
            out.writeObject(this.grid);
            out.writeObject(this.player);
            out.writeObject(this.aiPlayer);
            out.writeObject(this.entrance);
            out.writeObject(this.exit);
            out.writeInt(this.humanMoveCount);
            out.writeInt(this.aiMoveCount);
            out.writeInt(this.elapsedTime);
            System.out.println("Game saved successfully.");
        } catch (IOException e) {
            System.out.println("Error saving game: " + e.getMessag
```

```
        }

    }
```

- **Purpose**: Saves the current game state to a file (`savegame.dat`) using Java serialization.
- **Process**:

1. Opens an `ObjectOutputStream` to `savegame.dat`.

2. Writes essential game state data, including maze dimensions, grid, player positions, move counts, and elapsed time.

3. Closes the stream and confirms the save operation to the player.

`loadGame()`

```
public void loadGame() {

        try (ObjectInputStream in = new ObjectInputStream(new File

            this.width = in.readInt();

            this.height = in.readInt();

            this.cellSize = in.readInt();

            this.grid = (Cell[][]) in.readObject();

            this.player = (Player) in.readObject();

            this.aiPlayer = (Player) in.readObject();

            this.entrance = (Cell) in.readObject();

            this.exit = (Cell) in.readObject();

            this.humanMoveCount = in.readInt();

            this.aiMoveCount = in.readInt();

            this.elapsedTime = in.readInt();
```

```
            if (isCompetitiveMode) {
                aiSolution.clear();
                aiPlannedMoves.clear();
                findSolution(getCell(player.x, player.y), new Hash
                aiPlannedMoves.addAll(aiSolution);
            }


            System.out.println("Game loaded successfully.");
            isGameRunning = true;
            redrawMazeAndPlayer();
        } catch (IOException | ClassNotFoundException e) {
            System.out.println("Error loading game: " + e.getMessa
        }
    }
```

- **Purpose**: Loads a saved game state from `savegame.dat` using Java deserialization.
- **Process**:

1. Opens an `ObjectInputStream` from `savegame.dat`.

2. Reads and restores the game state data, including maze dimensions, grid, player positions, move counts, and elapsed time.

3. If in competitive mode, recalculates the AI's solution path.

4. Sets the game as running and redraws the maze and players.

5. Confirms the load operation to the player.

5.5. Game Management

`startGame()`

```
public void startGame() {

        humanMoveCount = 0;

        generate();

        draw();

        initializePlayer();

        startTimer();

        startRegenerationTimer();

        runGameLoop();

    }
```

- **Purpose**: Initiates a single-player game.
- **Process**:

1. Resets the human move count.
2. Calls `generate()` to create a new maze.
3. Calls `draw()` to render the maze.
4. Initializes the player at the entrance by calling `initializePlayer()`.
5. Starts the game timer with `startTimer()`.
6. Starts the maze regeneration timer with `startRegenerationTimer()`.
7. Enters the main game loop by calling `runGameLoop()`.

`startCompetitiveMode()`

```
public void startCompetitiveMode() {

        humanMoveCount = 0;
```

```
        generate();

        isCompetitiveMode = true;

        isHumanTurn = true;



        player = new Player(entrance);

        aiPlayer = new Player(entrance);



        // Pre-calculate AI solution so it has a path

        aiSolution.clear();

        aiPlannedMoves.clear();



        findSolution(getCell(player.x, player.y), new HashSet<>(),

        aiPlannedMoves.addAll(aiSolution);



        draw();

        redrawMazeAndPlayer();

        startTimer();

        startRegenerationTimer();

        runCompetitiveGameLoop();

    }
```

- **Purpose**: Initiates a competitive game where the player competes against an AI to reach the exit first.
- **Process**:

1. Resets the human move count.

2. Calls `generate()` to create a new maze.

3. Sets `isCompetitiveMode` to `true` and `isHumanTurn` to `true`.

4. Initializes both the human player and AI player at the entrance.

5. Clears any existing AI solutions and recalculates the AI's solution path using `findSolution()`.

6. Copies the AI's solution to `aiPlannedMoves`.

7. Calls `draw()` to render the maze.

8. Redraws the maze and players with `redrawMazeAndPlayer()`.

9. Starts the game timer with `startTimer()`.

10. Starts the maze regeneration timer with `startRegenerationTimer()`.

11. Enters the competitive game loop by calling `runCompetitiveGameLoop()`.

5.6. Game Initialization

play()

```
 public void play() {
          System.out.println("Welcome to the Maze Game!");
          System.out.println("1) Load Last Saved Game");
          System.out.println("2) Start New Game");

          int choice = -1;
          while (choice != 1 && choice != 2) {
              System.out.print("Enter your choice (1 or 2): ");
              choice = scanner.nextInt();
              scanner.nextLine(); // Consume newline
```

```
            }


            if (choice == 1) {
                // Attempt load
                loadGame();
                if (isGameRunning) {
                    System.out.println("Continuing the loaded game...'
                    continueGame();
                    return;
                } else {
                    System.out.println("No saved game found or error c
                }
            }


            // Choose difficulty
            System.out.println("Select difficulty:");
            System.out.println("1) Easy (10x10)
2) Medium (20x20)
3) Hard (30x30)");
            int diffChoice = scanner.nextInt();
            scanner.nextLine(); // consume leftover


            int size;
            switch (diffChoice) {
```

```
            case 2 -> size = 20; // Medium
            case 3 -> size = 30; // Hard
            default -> size = 10; // Easy
        }

        // We'll keep cellSize always 20px
        int cellSize = 20;

        // Re-initialize Maze with chosen settings
        this.width = size;
        this.height = size;
        this.cellSize = cellSize;
        this.grid = initializeGrid();

        // Choose mode
        System.out.println("Select game mode:");
        System.out.println("1) Single Player
2) Competitive (vs AI)");
        int mode = scanner.nextInt();
        scanner.nextLine(); // consume leftover

        if (mode == 2) {
            startCompetitiveMode();
```

```
        } else {
            startGame();
        }
    }
```

- **Purpose**: Serves as the main entry point for the game, handling user interactions for starting or loading games.
- **Process**:

1. Displays a welcome message and menu options:

- `1) Load Last Saved Game`

- `2) Start New Game`

2. Prompts the user to enter their choice (`1` or `2`) and validates the input.

3. If the user chooses to load a game (`1`):

- Calls `loadGame()`.

- If loading is successful and `isGameRunning` is `true`, calls `continueGame()` to resume.

- If loading fails, notifies the user and proceeds to start a new game.

4. If the user chooses to start a new game (`2`):

- Prompts the user to select a difficulty level:

- `1) Easy (10x10)`

- `2) Medium (20x20)`

- `3) Hard (30x30)`

- Reads and validates the difficulty choice, setting the maze size accordingly (`10`, `20`, or `30`).

- Prompts the user to choose a game mode:

- `1) Single Player`

- 2) Competitive (vs AI)

- Reads and validates the game mode choice.

- Initializes the maze grid with the chosen settings by calling `initializeGrid()`.

- Initializes the player by calling `initializePlayer()` if in Single Player mode.

- Starts the game by calling `startGame()` or `startCompetitiveMode()` based on the selected mode.

5.7. Game Loop Management

`runGameLoop()`

```
 private void runGameLoop() {

         while (isGameRunning) {

             System.out.print("Move (WASD/solve/next/save/load/q):

             String input = scanner.nextLine().toLowerCase();

             if (input.isEmpty()) continue;



             switch (input) {

                 case "q" -> {

                     stopGame();

                     return;

                 }

                 case "solve" -> solveMaze();

                 case "next" -> provideNext10Steps();

                 case "save" -> saveGame();

                 case "load" -> loadGame();
```

```
                default -> {
                    // If first char is in WASD
                    if ("wasd".indexOf(input.charAt(0)) != -1) {
                        movePlayer(input.charAt(0));
                    }
                }
            }
        }
        stopGame();
    }
```

- **Purpose**: Manages the main loop for single-player mode, handling user input and game progression.
- **Process**:

1. Continuously runs while `isGameRunning` is `true`.

2. Prompts the player for input commands:

- Movement commands: `W`, `A`, `S`, `D`

- Other commands: `solve`, `next`, `save`, `load`, `q`

3. Reads and processes the input:

- **`q`**: Quits the game by calling `stopGame()`.

- **`solve`**: Calls `solveMaze()` to display the full solution path.

- **`next`**: Calls `provideNext10Steps()` to display the next 10 steps of the solution.

- **`save`**: Calls `saveGame()` to save the current game state.

- **`load`**: Calls `loadGame()` to load a saved game state.

- **Movement Commands (`W`, `A`, `S`, `D`)**: Calls `movePlayer(char direction)` with the corresponding direction.

4. Continues the loop until the game is stopped or the player quits.

5.8. AI Path Execution

`makeAIMove()`

```
 private void makeAIMove() {
         if (!aiPlannedMoves.isEmpty()) {
             char move = aiPlannedMoves.remove(0);
             moveAIPlayer(move);
         }
     }
```

- **Purpose**: Executes the AI's next move based on its pre-planned solution path.
- **Process**:

1. Checks if `aiPlannedMoves` is not empty.

2. Retrieves the next move (`char`) from `aiPlannedMoves`.

3. Calls `moveAIPlayer(char direction)` to perform the AI's move.

4. Increments the AI's move count (`aiMoveCount`).

5. Checks if the AI has reached the exit by calling `checkAIWinCondition()`.

6. Switches the turn back to the human player by setting `isHumanTurn` to `true`.

7. Displays the current move counts for both players.

---

6. Helper Methods

Helper methods support primary methods by handling specific tasks, enhancing code modularity and readability.

6.1. Maze Generation Helpers

- **`resetGrid()`**

```
    private void resetGrid() {
        for (Cell[] row : grid) {
            for (Cell cell : row) {
                cell.visited = false;
                Arrays.fill(cell.walls, true);
            }
        }
    }
```

- **Purpose**: Resets the maze grid, marking all cells as unvisited and restoring all walls.

- **Process**:

- Iterates through each cell in the `grid` and sets `visited` to `false`.

- Restores all walls by setting each element in `cell.walls` to `true`.

- **`getUnvisitedNeighbor(Cell cell)`**

```
    private Cell getUnvisitedNeighbor(Cell cell) {
        List<Cell> neighbors = new ArrayList<>();
        int[][] directions = {{0, -1}, {1, 0}, {0, 1}, {-1, 0}};
        for (int[] dir : directions) {
            int newX = cell.x + dir[0];
            int newY = cell.y + dir[1];
            if (isValidCell(newX, newY) && !grid[newY][newX].visit
                neighbors.add(grid[newY][newX]);
            }
        }
```

```
        return neighbors.isEmpty() ? null : neighbors.get(random.r
    }
```

- **Purpose**: Finds an unvisited neighbor of the given cell.

- **Process**:

- Defines possible directions (Up, Right, Down, Left) with corresponding deltas.

- Iterates through each direction to check if the neighboring cell is within bounds and unvisited.

- Collects all unvisited neighbors and returns a randomly selected one.

- Returns `null` if no unvisited neighbors are found.

- **`removeWalls(Cell current, Cell next)`**

```
private void removeWalls(Cell current, Cell next) {
        int dx = next.x - current.x;
        int dy = next.y - current.y;

        if (dx == 1) { // right
            current.walls[1] = false;
            next.walls[3] = false;
        } else if (dx == -1) { // left
            current.walls[3] = false;
            next.walls[1] = false;
        } else if (dy == 1) { // down
            current.walls[2] = false;
            next.walls[0] = false;
```

```
        } else if (dy == -1) { // up
            current.walls[0] = false;
            next.walls[2] = false;
        }
    }
```

- **Purpose**: Removes walls between two adjacent cells to create a passage.

- **Process**:

- Calculates the difference in `x` and `y` coordinates between `next` and `current` cells.

- Determines the direction of movement based on coordinate differences.

- Removes the corresponding walls in both `current` and `next` cells.

- **`createEntranceAndExit()`**

```
private void createEntranceAndExit() {
        int entranceSide = random.nextInt(4);
        int exitSide;
        do {
            exitSide = random.nextInt(4);
        } while (exitSide == entranceSide);

        entrance = createOpeningOnSide(entranceSide);
        exit = createOpeningOnSide(exitSide);
    }
```

- **Purpose**: Creates an entrance and an exit on random sides of the maze.

- **Process**:

- Randomly selects a side for the entrance.

- Ensures the exit is created on a different side to maintain maze complexity.

- Calls `createOpeningOnSide(int side)` for both entrance and exit.

- **`createOpeningOnSide(int side)`**

```
private Cell createOpeningOnSide(int side) {
        Cell cell;
        switch (side) {
            case 0 -> { // Top
                cell = grid[0][random.nextInt(width)];
                cell.walls[0] = false;
            }
            case 1 -> { // Right
                cell = grid[random.nextInt(height)][width - 1];
                cell.walls[1] = false;
            }
            case 2 -> { // Bottom
                cell = grid[height - 1][random.nextInt(width)];
                cell.walls[2] = false;
            }
            case 3 -> { // Left
                cell = grid[random.nextInt(height)][0];
                cell.walls[3] = false;
            }
```

```
            default -> throw new IllegalArgumentException("Invalic
        }
        return cell;
    }
```

- **Purpose**: Creates an opening (entrance or exit) on a specified side of the maze.

- **Parameters**:

- `side`: An integer representing the maze side (0=Top, 1=Right, 2=Bottom, 3=Left).

- **Process**:

- Selects a random cell on the specified side.

- Removes the wall on that side to create an opening.

- Returns the modified `Cell` object.

6.2. Movement Handling

- **`movePlayer(char direction)`**

```
private void movePlayer(char direction) {
        if (player == null) return;
        int[] movement = getMovementDeltas(direction);
        int dx = movement[0], dy = movement[1];

        if (canMove(player.x, player.y, dx, dy)) {
            updatePlayerPosition(dx, dy);
            checkWinCondition();
            redrawMazeAndPlayer();
            humanMoveCount++;
```

```
        }

    }
```

- **Purpose**: Moves the human player in the specified WASD direction.

- **Parameters**:

- `direction`: A character representing the direction (`'w'`, `'a'`, `'s'`, `'d'`).

- **Process**:

1. Converts the direction to movement deltas `(dx, dy)` using `getMovementDeltas(char direction)`.

2. Checks if the move is valid using `canMove(int currentX, int currentY, int dx, int dy)`.

3. If valid, updates the player's position with `updatePlayerPosition(int dx, int dy)`.

4. Checks for a win condition using `checkWinCondition()`.

5. Redraws the maze and player with `redrawMazeAndPlayer()`.

6. Increments the human move count (`humanMoveCount`).

- **`getMovementDeltas(char direction)`**

```
private int[] getMovementDeltas(char direction) {

        return switch (direction) {

            case 'w' -> new int[]{0, -1};   // Up

            case 's' -> new int[]{0, 1};    // Down

            case 'a' -> new int[]{-1, 0};   // Left

            case 'd' -> new int[]{1, 0};    // Right

            default -> new int[]{0, 0};

        };
```

```
    }
```

- **Purpose**: Converts a WASD character into movement deltas `(dx, dy)`.

- **Parameters**:

- `direction`: A character representing the direction (`'w'`, `'a'`, `'s'`, `'d'`).

- **Returns**: An integer array `{dx, dy}` representing movement in the maze.

- **Mappings**:

- `'w'` → `{0, -1}` (Up)

- `'s'` → `{0, 1}` (Down)

- `'a'` → `{-1, 0}` (Left)

- `'d'` → `{1, 0}` (Right)

- Default → `{0, 0}` (No movement)

- **`canMove(int currentX, int currentY, int dx, int dy)`**

```
private boolean canMove(int currentX, int currentY, int dx, int dy
        if (!isValidCell(currentX + dx, currentY + dy)) {
            return false;
        }
        Cell currentCell = grid[currentY][currentX];
        return switch (getDirectionIndex(dx, dy)) {
            case 0 -> !currentCell.walls[0]; // Top
            case 1 -> !currentCell.walls[1]; // Right
            case 2 -> !currentCell.walls[2]; // Bottom
            case 3 -> !currentCell.walls[3]; // Left
            default -> false;
```

```
        };

    }
```

- **Purpose**: Determines if the player can move from `(currentX, currentY)` by `(dx, dy)`.

- **Parameters**:

- `currentX`, `currentY`: Current coordinates of the player.

- `dx`, `dy`: Movement deltas.

- **Returns**: `true` if the move is valid; `false` otherwise.

- **Process**:

1. Checks if the target cell `(currentX + dx, currentY + dy)` is within maze bounds using `isValidCell(int x, int y)`.

2. Retrieves the current cell from the grid.

3. Determines the direction index corresponding to `(dx, dy)` using `getDirectionIndex(int dx, int dy)`.

4. Checks if the wall in that direction is open (`false` in `cell.walls[direction]`).

- **`getDirectionIndex(int dx, int dy)`**

```
private int getDirectionIndex(int dx, int dy) {

        if (dy < 0) return 0;   // Top

        if (dx > 0) return 1;   // Right

        if (dy > 0) return 2;   // Bottom

        if (dx < 0) return 3;   // Left

        return -1;

    }
```

- **Purpose**: Converts movement deltas `(dx, dy)` into a direction index.

- **Parameters**:

- `dx`, `dy`: Movement deltas.

- **Returns**: An integer representing the direction (`0=Top`, `1=Right`, `2=Bottom`, `3=Left`, `-1=Invalid`).

- **Mappings**:

- `(0, -1)` → `0` (Top)

- `(1, 0)` → `1` (Right)

- `(0, 1)` → `2` (Bottom)

- `(-1, 0)` → `3` (Left)

- Others → `-1`

- **`updatePlayerPosition(int dx, int dy)`**

```
private void updatePlayerPosition(int dx, int dy) {
        player.x += dx;
        player.y += dy;
    }
```

- **Purpose**: Updates the player's coordinates based on movement deltas.

- **Parameters**:

- `dx`, `dy`: Movement deltas.

- **Process**:

- Adds `dx` to `player.x`.

- Adds `dy` to `player.y`.

- **`checkWinCondition()`**

```
private void checkWinCondition() {
        if (isCompetitiveMode) {
```

```
                if (player.x == exit.x && player.y == exit.y) {
                    int finalTime = elapsedTime;
                    redrawMazeAndPlayers();
                    stopGame();
                    System.out.println("Game Over! You win in " + huma
                    System.out.println("Time elapsed: " + finalTime +
                    System.out.println("AI moves: " + aiMoveCount);
                }
            } else {
                if (player.x == exit.x && player.y == exit.y) {
                    int finalTime = elapsedTime;
                    redrawMazeAndPlayer();
                    stopGame();
                    System.out.println("Congratulations! You completec
                }
            }
        }
```

- **Purpose**: Checks if the human player has reached the exit and handles game termination.

- **Process**:

1. Compares the player's current position with the exit's position.

2. If in competitive mode:

- If the player reaches the exit, stops the game and announces the player as the winner, displaying move counts and elapsed time.

3. If in single-player mode:

- If the player reaches the exit, stops the game and congratulates the player, displaying elapsed time.

6.3. Path Drawing

- **drawPathInGreen(Cell startCell, List<Character> moves)**

```
private void drawPathInGreen(Cell startCell, List<Character> moves

        // Temporarily store x,y

        int sx = startCell.x;

        int sy = startCell.y;



        // We re-draw the maze so we can place the path on top

        turtle.reset();

        draw();  // Draw the walls, entrance, exit, timer, etc.



        // If competitive, draw both players; otherwise, just the

        if (isCompetitiveMode) {

            drawBothPlayers();

        } else {

            drawPlayerPosition();

        }



        // Now draw the path

        turtle.color(PATH_COLOR);
```

```
            turtle.penDown();


            double xPos = sx * cellSize + cellSize / 2.0;
            double yPos = sy * cellSize + cellSize / 2.0;
            turtle.moveTo(xPos, yPos);


            for (char move : moves) {
                int[] delta = getMovementDeltas(move);
                sx += delta[0];
                sy += delta[1];


                double newX = sx * cellSize + cellSize / 2.0;
                double newY = sy * cellSize + cellSize / 2.0;
                turtle.lineTo(newX, newY);
            }
            turtle.penUp();
        }
    }
```

- **Purpose**: Visually represents a path on the maze using green lines from a specified starting cell.

- **Parameters**:

- `startCell`: The `Cell` object from which the path begins.

- `moves`: A list of `Character` representing the sequence of moves (`'w'`, `'a'`, `'s'`, `'d'`).

- Process:

1. Stores the starting coordinates (`sx`, `sy`).

2. Resets the Turtle and redraws the maze and players.

3. Sets the Turtle color to `PATH_COLOR` and initiates drawing.

4. Moves the Turtle to the starting position.

5. Iterates through the list of moves, updating coordinates and drawing lines to represent the path.

6. Lifts the Turtle pen after drawing.

6.4. Timer Management

- **`startTimer()`**

```
private void startTimer() {
        if (gameTimer != null) return;

        elapsedTime = 0;
        isGameRunning = true;

        gameTimer = new Timer();
        gameTimer.scheduleAtFixedRate(new TimerTask() {
            @Override
            public void run() {
                elapsedTime++;
            }
        }, 1000, 1000);
```

```
    }
```

- **Purpose**: Initiates the main game timer that tracks elapsed time.

- **Process**:

1. Checks if `gameTimer` is already running to prevent multiple timers.

2. Resets `elapsedTime` to `0` and sets `isGameRunning` to `true`.

3. Creates a new `Timer` instance.

4. Schedules a `TimerTask` to increment `elapsedTime` every second.

- **`stopGame()`**

```
private void stopGame() {
        isGameRunning = false;
        if (gameTimer != null) {
            gameTimer.cancel();
            gameTimer.purge();
            gameTimer = null;
        }
        if (displayTimer != null) {
            displayTimer.cancel();
            displayTimer.purge();
            displayTimer = null;
        }
        if (regenerationTimer != null) {
            regenerationTimer.cancel();
            regenerationTimer.purge();
```

```
            regenerationTimer = null;

        }

        aiSolution.clear();

        aiPlannedMoves.clear();

        aiMoveCount = 0;

        isCompetitiveMode = false;

    }
```

- **Purpose**: Stops all active timers and ends the game gracefully.

- **Process**:

1. Sets `isGameRunning` to `false`.

2. Cancels and purges `gameTimer`, `displayTimer`, and `regenerationTimer` if they are active.

3. Clears AI solution paths and resets move counts.

4. Resets competitive mode flags to default.

5. Notifies the player that the game has ended.

6.5. Utility Methods

- **`isValidCell(int x, int y)`**

```
private boolean isValidCell(int x, int y) {

        return x >= 0 && x < width && y >= 0 && y < height;

    }
```

- **Purpose**: Checks if the coordinates `(x, y)` are within the maze boundaries.

- **Parameters**:

- `x`, `y`: Coordinates to validate.

- **Returns**: `true` if within bounds; `false` otherwise.

- **`getCell(int x, int y)`**

```
private Cell getCell(int x, int y) {
        return grid[y][x];
    }
```

- **Purpose**: Retrieves the `Cell` object at the specified coordinates.

- **Parameters**:

- `x`, `y`: Coordinates of the desired cell.

- **Returns**: The `Cell` object at `(x, y)`.

- **`delay(int milliseconds)`**

```
private void delay(int milliseconds) {
        try {
            Thread.sleep(milliseconds);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
```

- **Purpose**: Pauses execution for a specified duration, aiding in visualization.

- **Parameters**:

- `milliseconds`: Duration to pause in milliseconds.

- **Process**:

- Uses `Thread.sleep()` to pause execution.

- Handles `InterruptedException` by resetting the thread's interrupt status.

- **`initializeGrid()`**

```
 private Cell[][] initializeGrid() {
         Cell[][] newGrid = new Cell[height][width];
         for (int y = 0; y < height; y++) {
             for (int x = 0; x < width; x++) {
                 newGrid[y][x] = new Cell(x, y);
             }
         }
         return newGrid;
     }
```

- **Purpose**: Sets up the maze grid by initializing all cells.

- **Process**:

1. Creates a new 2D array of `Cell` objects based on `height` and `width`.

2. Initializes each `Cell` with its respective coordinates.

3. Returns the fully initialized grid.

- **`initializePlayer()`**

```
 private void initializePlayer() {
         player = new Player(entrance);
         drawPlayerPosition();
     }
```

- **Purpose**: Places the human player at the entrance of the maze.

- **Process**:

1. Creates a new `Player` instance positioned at the `entrance` cell.

2. Calls `drawPlayerPosition()` to visually represent the player.

- **`redrawMazeAndPlayer()`**

```
private void redrawMazeAndPlayer() {
        draw();
        drawPlayerPosition();
    }
```

- **Purpose**: Redraws the maze and the player's position in single-player mode.

- **Process**:

1. Calls `draw()` to render the maze.

2. Calls `drawPlayerPosition()` to render the player.

- **`redrawMazeAndPlayers()`**

```
private void redrawMazeAndPlayers() {
        if (turtle != null) {
            turtle.reset();
            draw();
            if (isCompetitiveMode) {
                drawBothPlayers();
            } else {
                drawPlayerPosition();
            }
        }
    }
```

- **Purpose**: Redraws the maze along with both the human player and AI in competitive mode.

- **Process**:

1. Resets the Turtle.

2. Calls `draw()` to render the maze.

3. Calls `drawBothPlayers()` to render both players.

---

7. Supporting Classes

These classes are defined in separate files and provide foundational structures used by the `Maze` class.

7.1. Cell Class

```
import java.io.Serializable;



/**
 * Represents a single cell within the maze.
 */
public class Cell implements Serializable {


    public final int x, y;

    public boolean visited;

    public final boolean[] walls;  // top, right, bottom, left



    /**
     * Constructs a Cell at the specified coordinates with all wal
```

```
     *

     * @param x The x-coordinate of the cell.

     * @param y The y-coordinate of the cell.

     */

    public Cell(int x, int y) {

        this.x = x;

        this.y = y;

        this.visited = false;

        this.walls = new boolean[]{true, true, true, true};

    }

}
```

Description:

- **Fields:**

- `x`, `y`: Coordinates of the cell within the maze grid.

- `visited`: Boolean flag indicating whether the cell has been visited during maze generation or solving.

- `walls`: Boolean array representing the presence of walls in four directions:

- `walls[0]`: Top wall

- `walls[1]`: Right wall

- `walls[2]`: Bottom wall

- `walls[3]`: Left wall

- **Constructor:**

- Initializes the cell's coordinates and sets all walls to `true` (all walls intact).

- Sets `visited` to `false` by default.

7.2. Player Class

```
import java.io.Serializable;



/**
 * Represents a player in the maze.
 */
public class Player implements Serializable {


    public int x, y;



    /**
     * Constructs a Player at the specified entrance cell.
     *
     * @param entrance The `Cell` object representing the maze's e
     */
    public Player(Cell entrance) {
        this.x = entrance.x;
        this.y = entrance.y;
    }
}
```

Description:

- **Fields:**

- `x`, `y`: Current coordinates of the player within the maze grid.

- **Constructor:**

- Initializes the player's position to the coordinates of the provided `entrance` cell.

7.3. Point Class

```
import java.io.Serializable;



/**
 * Represents a point in 2D space.
 */
public class Point implements Serializable {


    public final int x, y;


    /**
     * Constructs a Point at the specified coordinates.
     *
     * @param x The x-coordinate.
     * @param y The y-coordinate.
     */
    public Point(int x, int y) {
```

```
            this.x = x;

            this.y = y;

        }

    }
```

Description:

- **Fields:**

- `x`, `y`: Coordinates representing a point in 2D space.

- **Constructor:**

- Initializes the point's coordinates.

---

8. Summary of Methods

The following table provides an at-a-glance reference of all methods within the `Maze` class, categorized by their functionality and whether they are primary or helper methods.

Functionality	**Primary Methods**	**Helper Methods**
Maze Generation	`generate()`, `regenerateMaze()`	`resetGrid()`, `getUnvisitedNeighbor()`, `removeWalls()`, `createEntranceAndExit()`, `createOpeningOnSide()`
AI Solving	`findSolution()`, `solveMaze()`, `provideNext10Steps()`	`getDeltaFromDirection()`, `getCharFromDirection()`
Game Management	`startGame()`, `startCompetitiveMode()`	`runGameLoop()`, `runCompetitiveGameLoop()`, `makeAIMove()`

| | |

| **Game Initialization** | `play()` | *Uses* `initializeGrid()`, `initializePlayer()` |

| **Game Loop Management** | `runGameLoop()` | *None* |

| **AI Path Execution** | `makeAIMove()` | `moveAIPlayer()`, `updateAIPosition()`, `checkAIWinCondition()` |

| **Save/Load Game** | `saveGame()`, `loadGame()` | *None* |

| **Rendering** | `draw()`, `redrawMazeAndPlayer()`, `redrawMazeAndPlayers()`, `drawPathInGreen()` | `setupTurtle()`, `drawMazeStructure()`, `drawOuterBorder()`, `drawCellWalls()`, `drawWall()`, `drawPlayerPosition()`, `drawBothPlayers()`, `colorEntranceAndExit()`, `drawCellOpening()`, `drawTimer()`, `formatTime()` |

| **Movement Handling** | `movePlayer()` | `getMovementDeltas()`, `canMove()`, `getDirectionIndex()`, `updatePlayerPosition()`, `checkWinCondition()` |

| **Timer Management** | `startTimer()`, `stopGame()` | *None* |

| **Path Drawing** | `drawPathInGreen()` | *None* |

| **Utility Methods** | *None* | `isValidCell()`, `getCell()`, `delay()`, `initializeGrid()`, `initializePlayer()` |

III. Examples of Game Scenarios

Scenario 1: Single-Player Mode

1. **Starting the Game:**

- The player launches the Maze Game and selects **Start New Game**.

- Chooses **Easy (10x10)** difficulty.

2. **Navigating the Maze:**

- The maze is generated and displayed.

- The player uses `W`, `A`, `S`, `D` keys to move up, left, down, or right.

- If stuck, the player types `solve` to view the full solution path or `next` to see the next 10 steps.

3. **Winning the Game:**

- Upon reaching the exit, the game congratulates the player.

- Displays the total time taken and the number of moves made.

Scenario 2: Competitive Mode

1. **Starting the Game:**

- The player launches the Maze Game and selects **Competitive Mode**.

- Chooses **Medium (20x20)** difficulty.

2. **Gameplay:**

- Both the player and AI start at the maze's entrance.

- The AI computes its path to the exit using DFS.

- The game alternates turns between the player and AI:

- **Player's Turn**: Uses `W`, `A`, `S`, `D` to move.

- **AI's Turn**: Automatically moves based on its solution path.

3. **Winning the Game:**

- The first to reach the exit wins.

- The game announces the winner and provides statistics on moves and elapsed time.

Scenario 3: Save and Load

1. **Saving the Game:**

- During gameplay, the player decides to save progress.

- Types `save`, and the game state is written to `savegame.dat`.

2. **Loading the Game:**

- Later, the player resumes by selecting **Load Last Saved Game**.

- The game restores the maze, player positions, move counts, and elapsed time from `savegame.dat`.

3. **Continuing Gameplay:**

- The player continues navigating the maze from where they left off.

---

11. Conclusion

This comprehensive documentation provides a structured and detailed overview of the Maze Game's architecture, functionalities, and operational flow. It encompasses all primary and helper methods within the `Maze` class, as well as the supporting `Cell`, `Player`, and `Point` classes.

```
    public void generate() {
        resetGrid();
        Cell startCell = grid[0][0];
        startCell.visited = true;

        List<Cell> stack = new ArrayList<>();
```

```
        stack.add(startCell);



        while (!stack.isEmpty()) {

            Cell current = stack.get(stack.size() - 1);

            Cell next = getUnvisitedNeighbor(current);



            if (next != null) {

                removeWalls(current, next);

                next.visited = true;

                stack.add(next);

            } else {

                stack.remove(stack.size() - 1);

            }

        }

        createEntranceAndExit();

    }
```