جامعة حلوان HELWAN UNIVERSITY	Operating Systems 2 (Fall 2023) Project Discussion كلية الحاسبات والذكاء الإصطناعي Faculty of Computers & Artificial Intelligence

Operating systems 2- Rat in a maze project documentation

Project description :

Key Features:

1. **Mouse Movement Constraints:**
    - The algorithm simulates a mouse that can only move forward or downward in the maze. This constraint adds a layer of challenge and decision-making to the traversal process.
2. **Parallelization with Threads:**
    - At each decision point, the algorithm generates two possible directions. It continues the traversal in one direction and creates a new thread to explore the second direction simultaneously. This parallelized approach enhances efficiency and speed in maze exploration.
3. **Thread Limitation for Resource Optimization:**
    - The number of threads generated is limited based on the available processors. This limitation ensures optimal resource utilization, preventing excessive parallelization and maintaining a balance between speed and efficiency.
4. **Graphical User Interface (GUI):**
    - The project includes a user-friendly GUI that takes input for the maze size (N). The GUI dynamically updates and displays the maze in real-time as the algorithm progresses. Users can observe the virtual mouse's movements and the evolving path.
5. **Real-Time Maze Visualization:**
    - The maze visualization is continuously updated in real-time, providing an interactive and immersive experience for users. As the algorithm explores different paths, the GUI dynamically reflects the mouse's movements and the discovered path.

What the project is supposed to do :

A Maze is given as NXN binary matrix of blocks where source block is the upper left most block i.e, maze [0][0] and destination block is lower rightmost block i.e.,

maze[N-1][N-1]. A rat starts from source

and must reach the destination The rat can move only in two directions: forward and down. In the maze

matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to

destination. Use Multi-threading to solve this problem

You should design a multithreaded JAVA program with the following features:

- You should enter the dimensions of the maze, then a grid is generated.

- You should use the grid to specify dead blocks on runtime

Project Objectives:

- Develop a maze traversal algorithm with forward and downward movement constraints.
- Implement parallelization using threads for efficient exploration.
- Limit the number of threads based on the available processors to optimize resource usage.
- Design a user-friendly GUI for inputting the maze size and visualizing real-time maze traversal.
- Ensure the real-time update of the maze visualization to provide an interactive user experience.

Potential Extensions:

- Introduce maze generation algorithms to create dynamic and varied maze layouts.
- Implement additional constraints or challenges within the maze, such as obstacles or dead-ends.
- Explore different visualization styles or effects to enhance user engagement.
- Include performance metrics to measure the efficiency of parallelized maze traversal.

Expected Outcomes:

- A functional maze traversal algorithm with real-time visualization.
- An interactive GUI that allows users to input maze size and observe the mouse's path.
- Documentation outlining the project's architecture, algorithms, and usage instructions.

Operating systems 2- Rate in a Maze project documentation

Code Documentation

The provided Java code is for solving a maze using multithreading. Let's break down the key components and functionalities:

1. Multithreading Setup:

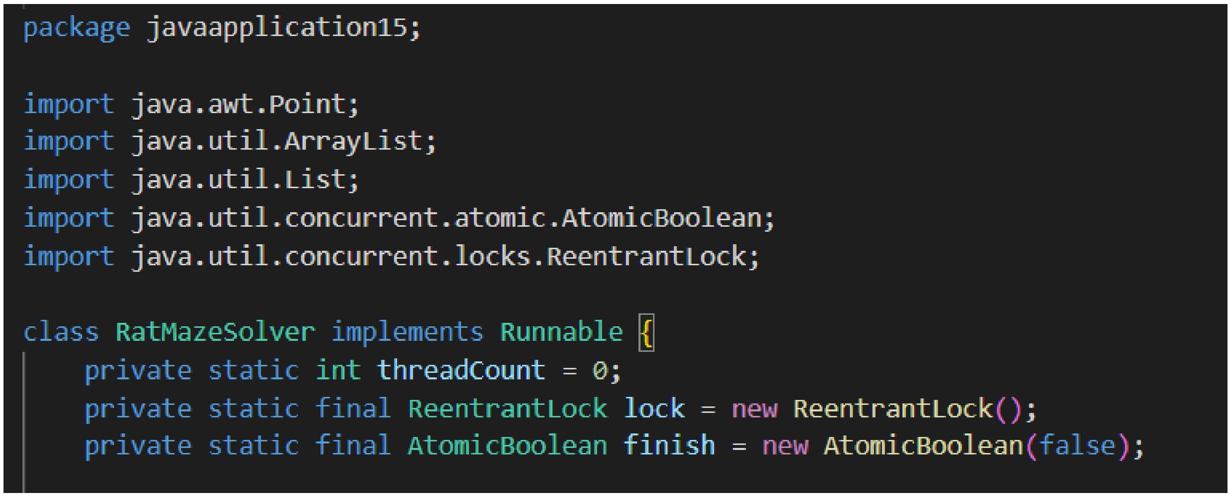
- The **RatMazeSolver** class implements the (Runnable) interface, indicating that instances of this class can be executed in separate threads.

- It includes static variables:

  - **threadCount**: Tracks the number of threads created.

  - lock: A **ReentrantLock** used to ensure thread-safe increment of **threadCount**.

  - finish: An **AtomicBoolean** used to signal the completion of maze-solving.

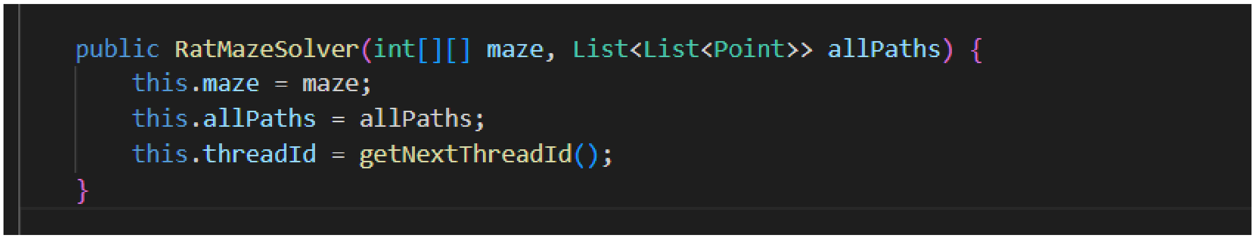


```
package javaapplication15;

import java.awt.Point;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.locks.ReentrantLock;

class RatMazeSolver implements Runnable {
    private static int threadCount = 0;
    private static final ReentrantLock lock = new ReentrantLock();
    private static final AtomicBoolean finish = new AtomicBoolean(false);
```

2. Constructor:

- The constructor of **RatMazeSolver** takes a 2D array **maze** representing the maze structure and a **List<List<Point>> allPaths** to store all possible paths.

- It initializes the **threadId** using the **getNextThreadId** method.

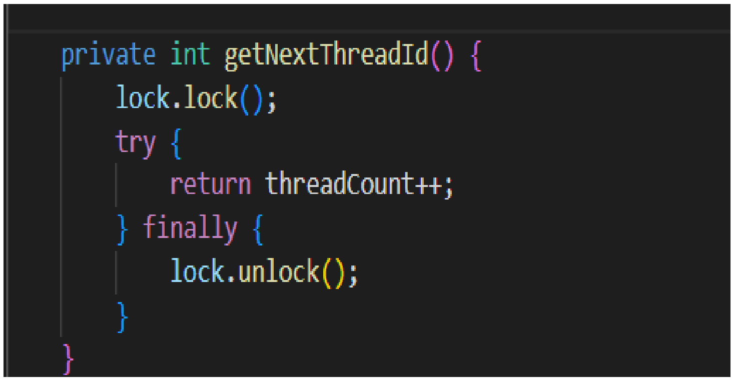


```java
public RatMazeSolver(int[][] maze, List<List<Point>> allPaths) {
    this.maze = maze;
    this.allPaths = allPaths;
    this.threadId = getNextThreadId();
}
```

3. GetNextThreadId Method:

- Ensures thread-safe increment of **threadCount** using the **ReentrantLock**.

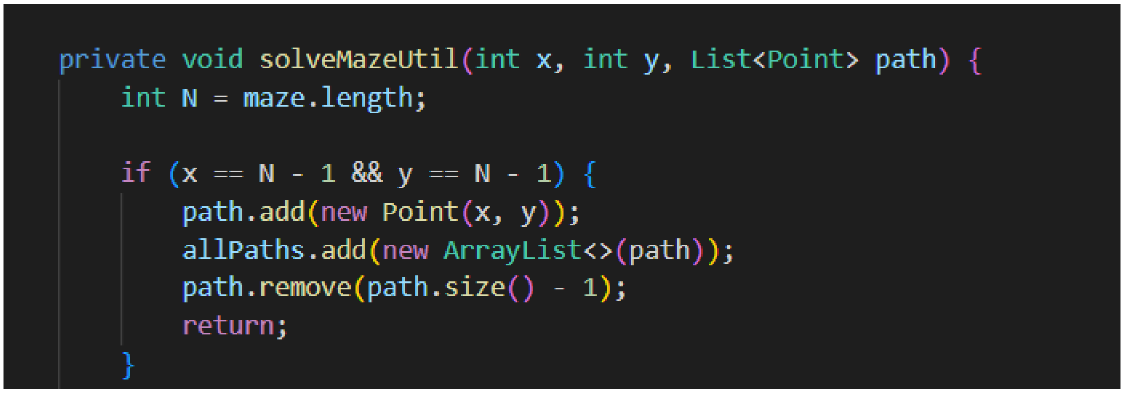
- Returns the current thread's unique identifier.



```java
private int getNextThreadId() {
    lock.lock();
    try {
        return threadCount++;
    } finally {
        lock.unlock();
    }
}
```

4. solveMazeUtil Method:

- Recursive method for exploring paths in the maze.
- Takes current coordinates (`x`, `y`) and a **List<Point>** representing the current path.
- Base case: If the current coordinates are at the destination, add the path to **allPaths**.
- Recursive exploration: Move down and move right, following valid paths.
- Backtracking: Remove the last point if no valid moves are possible from the current position.

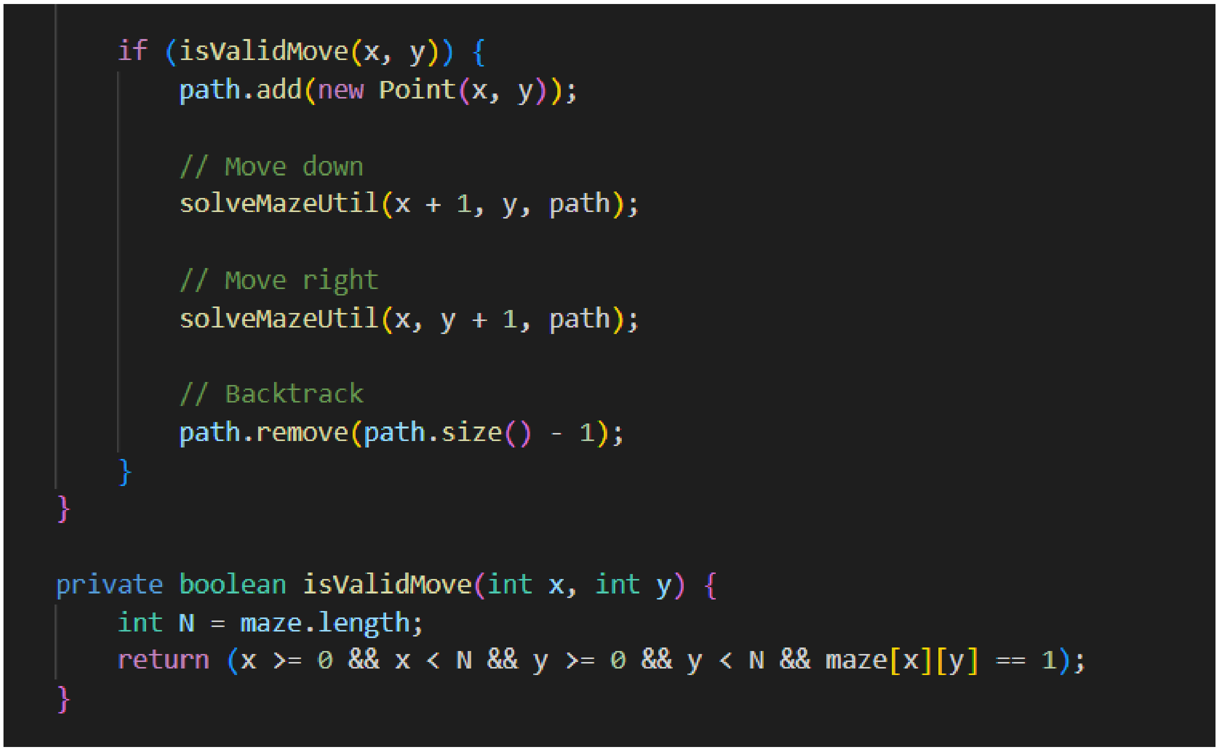


```java
private void solveMazeUtil(int x, int y, List<Point> path) {
    int N = maze.length;

    if (x == N - 1 && y == N - 1) {
        path.add(new Point(x, y));
        allPaths.add(new ArrayList<>(path));
        path.remove(path.size() - 1);
        return;
    }
```

5. isValidMove Method:

- Checks if a move to the specified coordinates (`x`, `y`) is valid.

- Ensures the coordinates are within the maze boundaries and the cell is accessible (contains `1`).



```
    if (isValidMove(x, y)) {
        path.add(new Point(x, y));

        // Move down
        solveMazeUtil(x + 1, y, path);

        // Move right
        solveMazeUtil(x, y + 1, path);

        // Backtrack
        path.remove(path.size() - 1);
    }
}

private boolean isValidMove(int x, int y) {
    int N = maze.length;
    return (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1);
}
```

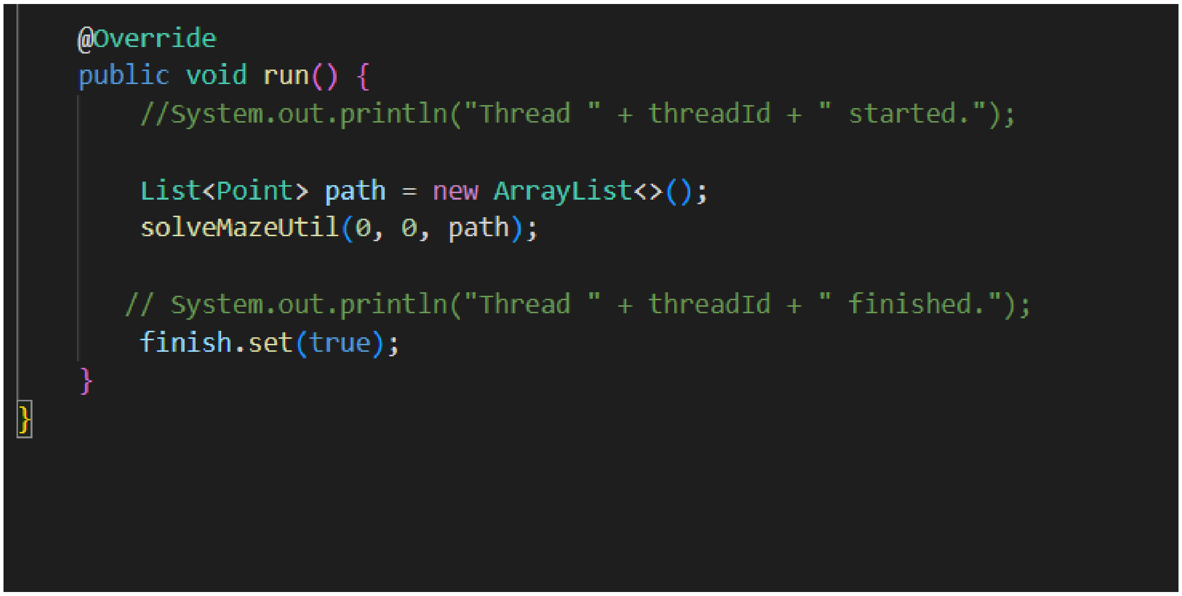
6. run Method:

- Overrides the **run** method from the **Runnable** interface.

- Initiates the maze-solving process for a single thread.

- Creates an empty path list and starts the recursive exploration from the top-left corner (start of the maze).

- Sets the **finish** flag to signal completion.



```
    @Override
    public void run() {
        //System.out.println("Thread " + threadId + " started.");

        List<Point> path = new ArrayList<>();
        solveMazeUtil(0, 0, path);

       // System.out.println("Thread " + threadId + " finished.");
        finish.set(true);
    }
}
```

Overall Explanation:

- The **`RatMazeSolver`** class is designed to be executed in a multithreaded environment, and each instance represents a thread solving the maze.

- It uses **recursion** and **backtracking** to explore all **possible paths** in the maze.

- The maze-solving is parallelized by creating **multiple threads**, each responsible for finding paths independently.

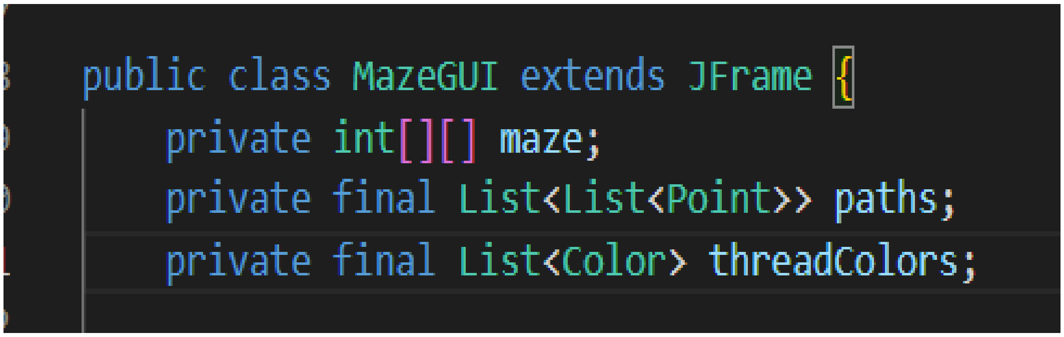
- The **`allPaths`** list collects all the possible paths explored by different threads.

This Java code defines a graphical user interface (GUI) for visualizing maze solving using multithreading. It includes a maze, paths found by multiple threads, and interactive features. Here's a breakdown of the code:

1. Class Structure:

- The **MazeGUI** class extends **JFrame** and serves as the main GUI for displaying the maze and paths.

- It includes member variables for the maze structure **(maze),** a list of paths **(paths),** and thread colors **(threadColors).**

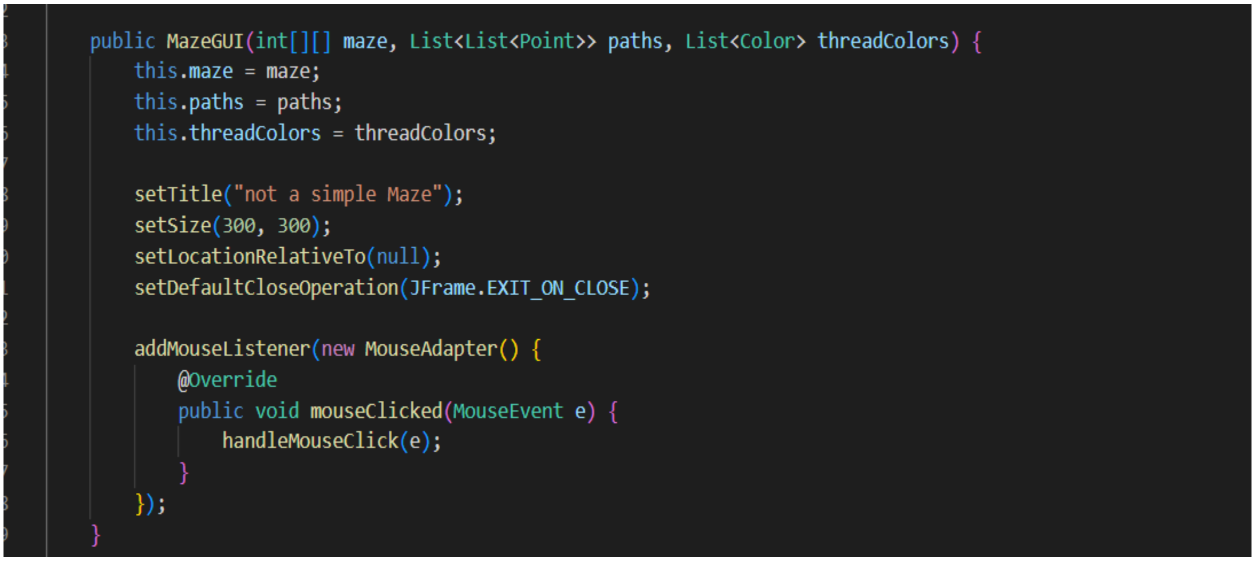


```
public class MazeGUI extends JFrame {
    private int[][] maze;
    private final List<List<Point>> paths;
    private final List<Color> threadColors;
```

2. Constructor:

- The constructor initializes the GUI settings, sets the title, size, and default close operation.

- It adds a **MouseListener** to handle mouse clicks.



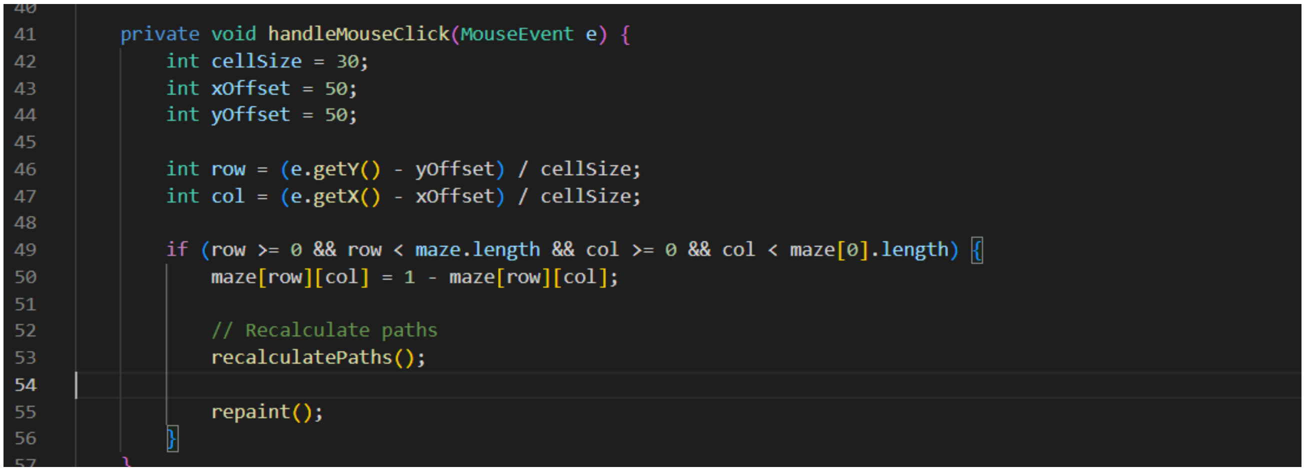
```java
public MazeGUI(int[][] maze, List<List<Point>> paths, List<Color> threadColors) {
    this.maze = maze;
    this.paths = paths;
    this.threadColors = threadColors;

    setTitle("not a simple Maze");
    setSize(300, 300);
    setLocationRelativeTo(null);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    addMouseListener(new MouseAdapter() {
        @Override
        public void mouseClicked(MouseEvent e) {
            handleMouseClick(e);
        }
    });
}
```

3. Mouse Click Handling (handleMouseClick method):

- Determines the cell clicked based on the mouse coordinates.

- Updates the maze by toggling the value of the clicked cell (0 to 1 or 1 to 0).

- Calls **recalculatePaths** to update the paths and triggers a repaint.



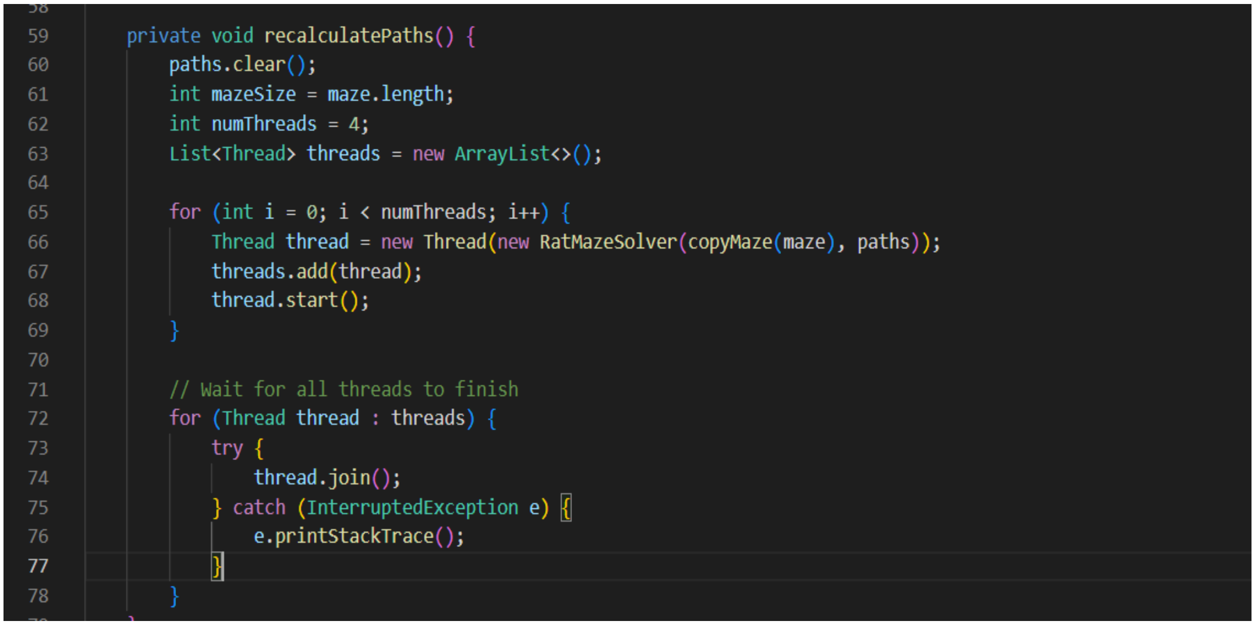
```java
    private void handleMouseClick(MouseEvent e) {
        int cellSize = 30;
        int xOffset = 50;
        int yOffset = 50;

        int row = (e.getY() - yOffset) / cellSize;
        int col = (e.getX() - xOffset) / cellSize;

        if (row >= 0 && row < maze.length && col >= 0 && col < maze[0].length) {
            maze[row][col] = 1 - maze[row][col];

            // Recalculate paths
            recalculatePaths();

            repaint();
        }
    }
```

4. Path Recalculation (recalculatePaths method):

- Clears the existing paths.

- Creates multiple threads (defined by **numThreads**) of **RatMazeSolver** instances to find paths concurrently.

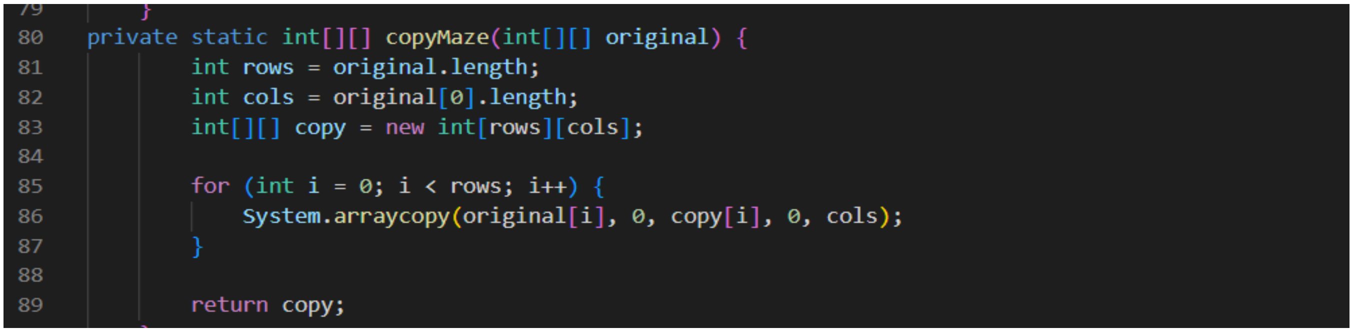
- Waits for all threads to finish using **thread.join().**



```java
    private void recalculatePaths() {
        paths.clear();
        int mazeSize = maze.length;
        int numThreads = 4;
        List<Thread> threads = new ArrayList<>();

        for (int i = 0; i < numThreads; i++) {
            Thread thread = new Thread(new RatMazeSolver(copyMaze(maze), paths));
            threads.add(thread);
            thread.start();
        }

        // Wait for all threads to finish
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

5. Copying Maze (copyMaze method):

- Creates a copy of the maze to be used by each thread independently.



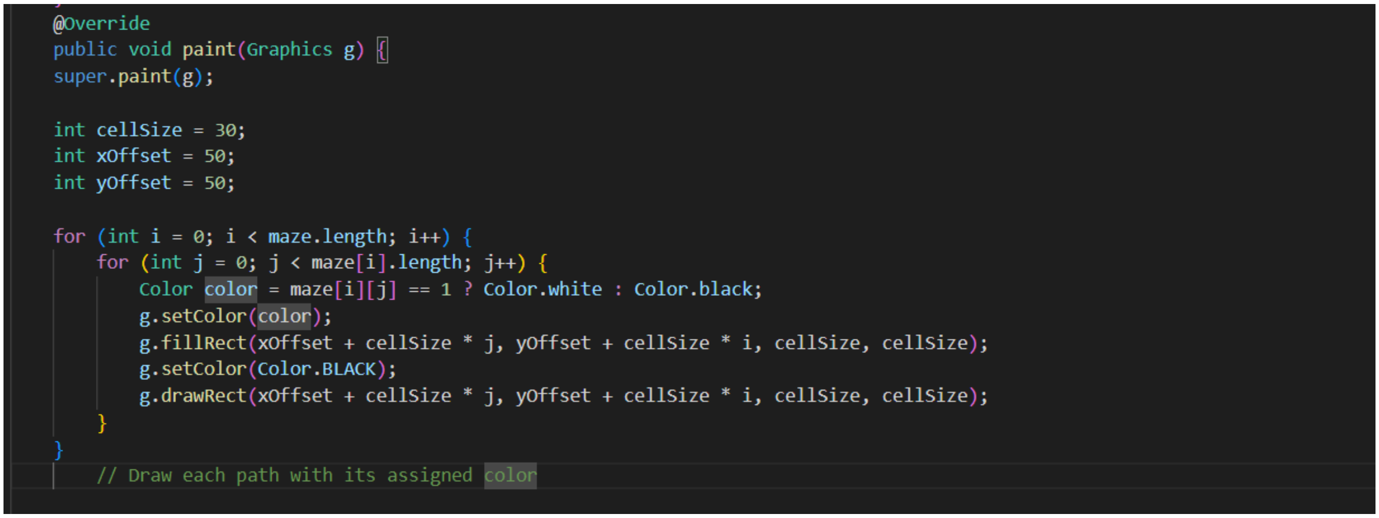
```
    }
private static int[][] copyMaze(int[][] original) {
        int rows = original.length;
        int cols = original[0].length;
        int[][] copy = new int[rows][cols];

        for (int i = 0; i < rows; i++) {
            System.arraycopy(original[i], 0, copy[i], 0, cols);
        }

        return copy;
```

6. paint Method (Override):

- Overrides the `paint` method to draw the maze, cells, and paths.

- Draws each cell of the maze based on its value (0 or 1).

- Draws each path with its assigned color.



```
@Override
public void paint(Graphics g) {
super.paint(g);

int cellSize = 30;
int xOffset = 50;
int yOffset = 50;

for (int i = 0; i < maze.length; i++) {
    for (int j = 0; j < maze[i].length; j++) {
        Color color = maze[i][j] == 1 ? Color.white : Color.black;
        g.setColor(color);
        g.fillRect(xOffset + cellSize * j, yOffset + cellSize * i, cellSize, cellSize);
        g.setColor(Color.BLACK);
        g.drawRect(xOffset + cellSize * j, yOffset + cellSize * i, cellSize, cellSize);
    }
}
    // Draw each path with its assigned color
```

7. Main Method:

- Asks the user to input the maze size using a dialog.

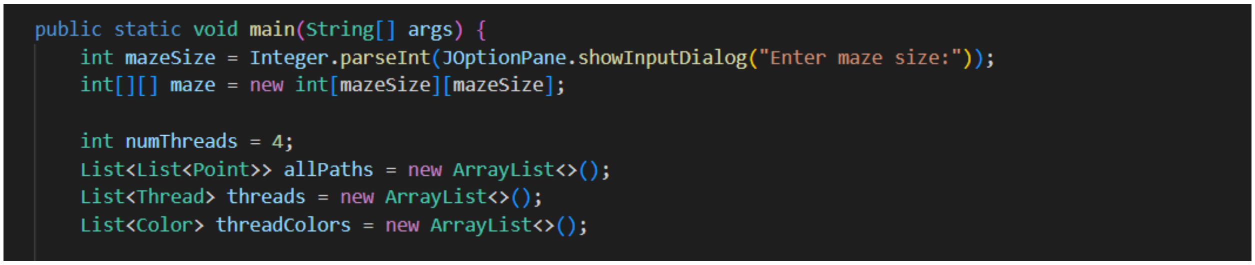
- Initializes maze, paths, threads, and thread colors.

- Creates an instance of **MazeGUI** and makes it visible.

- Starts threads to find paths concurrently.

- Waits for all threads to finish.

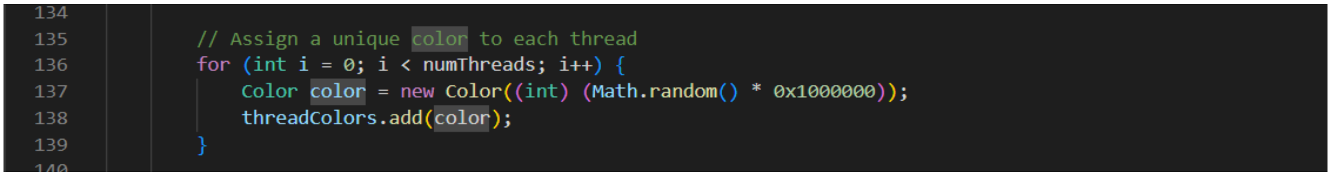
- Recalculates paths, triggers a repaint, and updates the GUI.



```java
public static void main(String[] args) {
    int mazeSize = Integer.parseInt(JOptionPane.showInputDialog("Enter maze size:"));
    int[][] maze = new int[mazeSize][mazeSize];

    int numThreads = 4;
    List<List<Point>> allPaths = new ArrayList<>();
    List<Thread> threads = new ArrayList<>();
    List<Color> threadColors = new ArrayList<>();
```

8. Thread Colors:

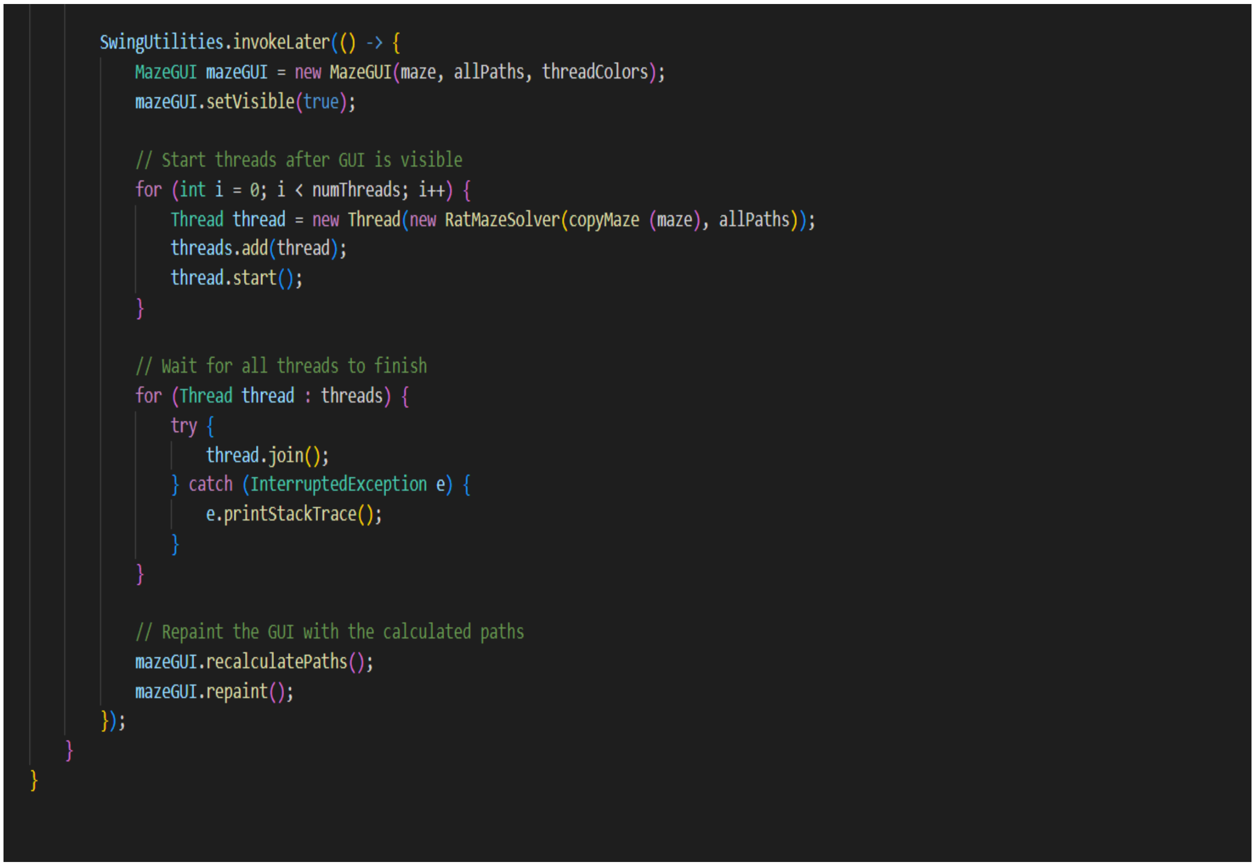
- Each thread is assigned a unique color, enhancing the visual distinction of paths.



```java
        // Assign a unique color to each thread
        for (int i = 0; i < numThreads; i++) {
            Color color = new Color((int) (Math.random() * 0x1000000));
            threadColors.add(color);
        }
```

9. SwingUtilities Invoked Later:

- Ensures that GUI-related tasks are performed in the event dispatch thread.



```
        SwingUtilities.invokeLater(() -> {
            MazeGUI mazeGUI = new MazeGUI(maze, allPaths, threadColors);
            mazeGUI.setVisible(true);

            // Start threads after GUI is visible
            for (int i = 0; i < numThreads; i++) {
                Thread thread = new Thread(new RatMazeSolver(copyMaze (maze), allPaths));
                threads.add(thread);
                thread.start();
            }

            // Wait for all threads to finish
            for (Thread thread : threads) {
                try {
                    thread.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            // Repaint the GUI with the calculated paths
            mazeGUI.recalculatePaths();
            mazeGUI.repaint();
        });
    }
}
```

*Note:

- This code combines GUI elements, multithreading, and maze-solving algorithms to create an interactive maze-solving visualization.

- The maze is displayed on the GUI, and as paths are found by multiple threads, they are dynamically updated and colored.

- The use of SwingUtilities ensures proper threading for GUI updates.

- The `**RatMazeSolver**` class (provided earlier) is used for maze-solving logic within each thread.

This project provides a visually engaging way to observe maze-solving algorithms in action and demonstrates the use of multithreading for parallel pathfinding.

Team members roles :

Multi-threading tasks:

Dalia Mahmoud -20210307

Hazem Emad -202102251

Habiba Sherif - 20210276

Jamal Sayed - 20210251

GUI :

Dalia Mahmoud - 20210307

Youssef Tarek - 20211078

Youssef Mohamed – 20211101

Documenations :

Habiba Sherif -20210267

Jamal Sayed - 20210251