ACKNOWLEDGEMENT

We are greatly indebted to our mini project guide **Ms.Nitu L Pariyal** for her able guidance throughout this work. It has been an altogether different experience to work with her and we would like to thank her for her help, suggestions and numerous discussions.

We gladly take this opportunity to thank **Dr.Mrs. Rajurkar A. M.** (Head of Computer Science & Engineering, MGM' s College of Engineering, Nanded).

We are heartily thankful to **Dr. Mrs. Lathkar G. S.** (Director, MGM's College of Engineering, Nanded) for providing facility during progress of mini project in java, also for her kindly help, guidance and inspiration. Last but not least we are also thankful to all those who help directly or indirectly to develop this mini project and complete it successfully.

With Deep Reverence,

Karn Deshmukh

ABSTRACT

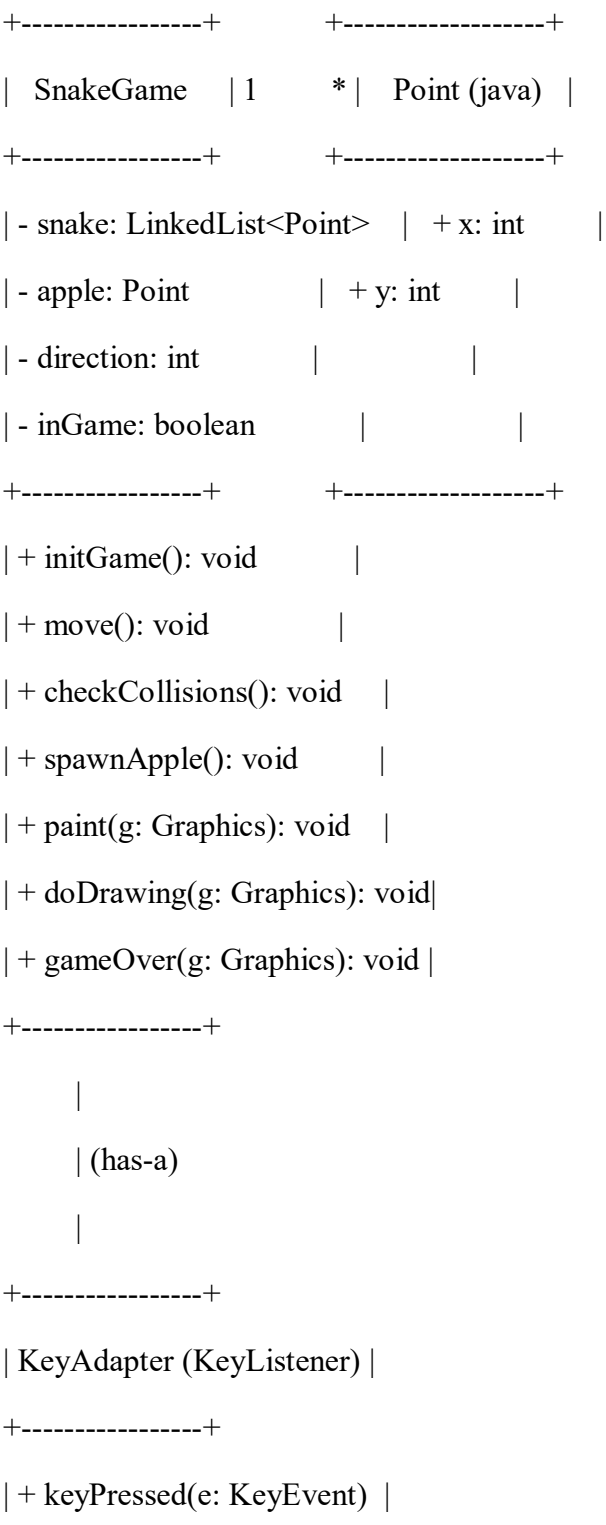
Snake Game Implementation in Java

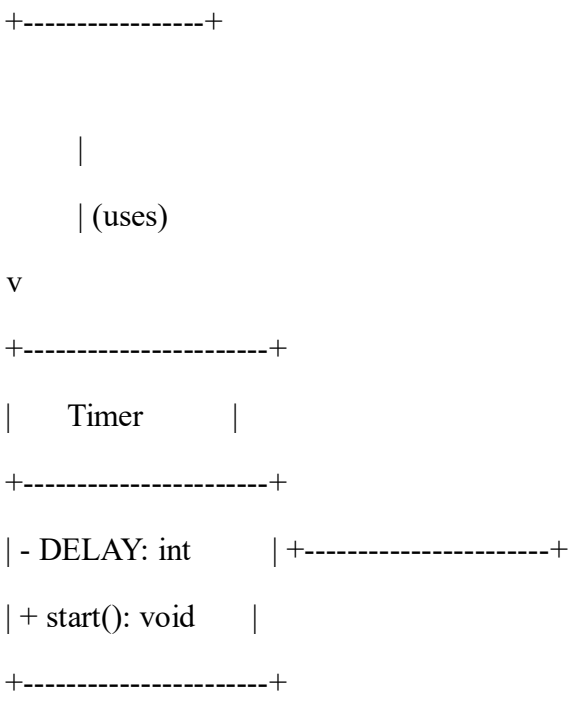
This paper presents a simple implementation of the classic Snake game using Java programming language. The game is designed to run on a graphical user interface (GUI) utilizing the javax.swing and java.awt libraries for rendering and user interaction. The primary objective of the game is to control a snake that moves on a grid, collecting food (apple) while avoiding collisions with walls or its own body. The game concludes when the snake collides with a boundary or itself. Event-Driven Programming: The game utilizes key event listeners to handle user input for controlling the snake's direction. The player interacts with the game through keyboard input, which is processed asynchronously. This simple Snake game in Java provides a foundational

understanding of game development principles such as game loops, collision detection, and event handling in graphical applications.

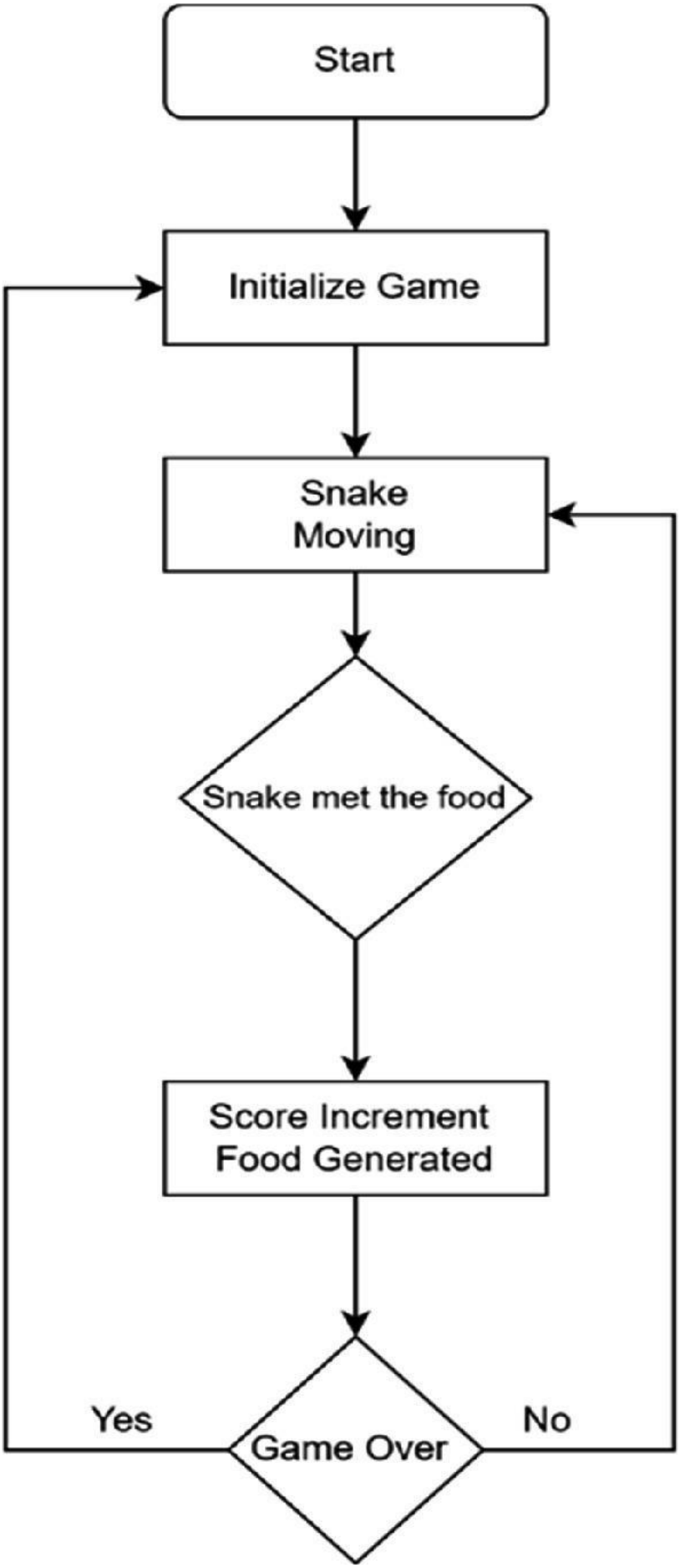
	TABLE OF CONTENT	
CHAPTER NO	**TITLE**	**PAGE NO**
	ACKNOWLEDGEMENT	**I**
	ABSTRACT	**II**
	TABLE OF CONTENT	**III**
1	UML Class Diagram	4
2	Flowchart	6
3	Code Of Snake Game	7
4	OUTPUT	11
5	Explanation Of Code	12
	CONCLUSION	18

1.UML Class Diagram




+----------+
|
| (uses)
v
+----------+
| Timer |
+----------+
| - DELAY: int |
| + start(): void |
+----------+

2.Flowchart


Start
Initialize Game
Snake
Moving
Snake met the food
Score Increment
Food Generated
Yes
Game Over
No

3.Code Of Snake Game

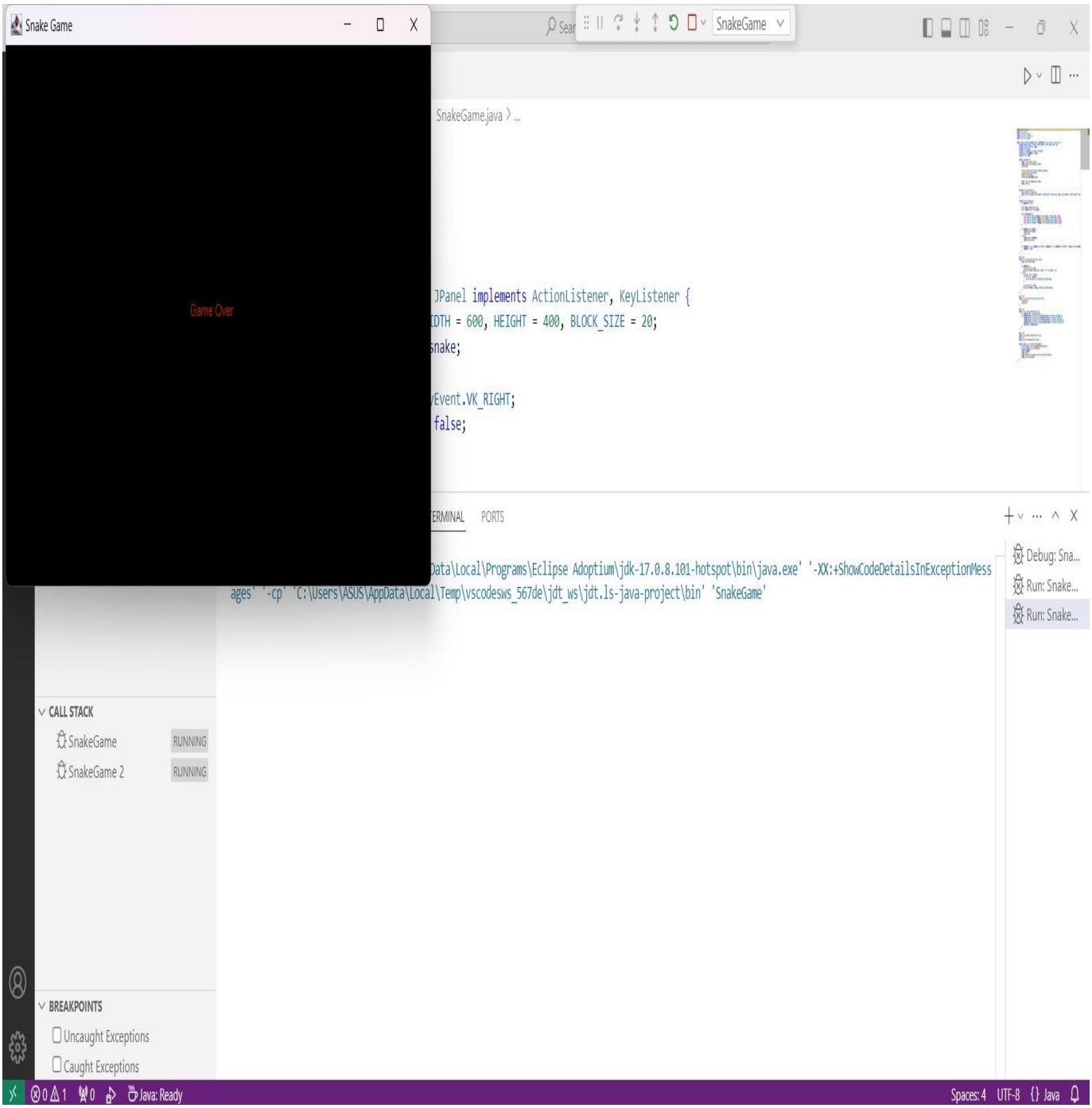
```
import javax.swing.*;
import java.awt.*; import
java.awt.event.*; import
java.util.LinkedList; import
java.util.Random;

public class SnakeGame extends JPanel implements ActionListener, KeyListener {
private static final int WIDTH = 600, HEIGHT = 400, BLOCK_SIZE = 20;
private LinkedList<Point> snake;     private Point food;
   private int direction = KeyEvent.VK_RIGHT;
private boolean gameOver = false;     private
Timer timer;

   public SnakeGame() {        snake
= new LinkedList<>();
snake.add(new Point(100, 100));
spawnFood();

       setPreferredSize(new Dimension(WIDTH, HEIGHT));
setBackground(Color.BLACK);
       addKeyListener(this);
setFocusable(true);

       timer = new Timer(100, this);
       timer.start();
   }

   private void spawnFood() {
       Random rand = new Random();
```

```
        food = new Point(rand.nextInt(WIDTH / BLOCK_SIZE) * BLOCK_SIZE, rand.nextInt(HEIGHT / BLOCK_SIZE) * BLOCK_SIZE);
    }

    private void moveSnake() {
if (gameOver) return;

        Point head = snake.getFirst();
        Point newHead = new Point(head);

        switch (direction) {             case KeyEvent.VK_UP: newHead.translate(0, -BLOCK_SIZE); break;             case KeyEvent.VK_DOWN: newHead.translate(0, BLOCK_SIZE); break;             case KeyEvent.VK_LEFT: newHead.translate(-BLOCK_SIZE, 0); break;             case KeyEvent.VK_RIGHT: newHead.translate(BLOCK_SIZE, 0); break;
        }

        if (newHead.equals(food)) {
snake.addFirst(food);             spawnFood();
        } else {
snake.addFirst(newHead);
snake.removeLast();
        }

        if (newHead.x < 0 || newHead.x >= WIDTH || newHead.y < 0 || newHead.y >= HEIGHT || snake.contains(newHead)) {
            gameOver = true;
        }
    }
```

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

        if (gameOver) {
            g.setColor(Color.RED);
            g.drawString("Game Over", WIDTH / 2 - 40, HEIGHT / 2);
        } else {
            g.setColor(Color.GREEN);
            for (Point p : snake) {
                g.fillRect(p.x, p.y, BLOCK_SIZE, BLOCK_SIZE);
            }

            g.setColor(Color.RED);
            g.fillRect(food.x, food.y, BLOCK_SIZE, BLOCK_SIZE);
        }
    }

    @Override     public void
actionPerformed(ActionEvent e) {
moveSnake();
        repaint();
    }

    @Override     public void keyPressed(KeyEvent e) {         int newDirection =
e.getKeyCode();         if ((newDirection == KeyEvent.VK_UP && direction !=
KeyEvent.VK_DOWN) ||             (newDirection == KeyEvent.VK_DOWN &&
direction != KeyEvent.VK_UP) ||
            (newDirection == KeyEvent.VK_LEFT && direction != KeyEvent.VK_RIGHT) ||
(newDirection == KeyEvent.VK_RIGHT && direction != KeyEvent.VK_LEFT)) {
            direction = newDirection;
```

```
        }
    }

    @Override     public void
keyReleased(KeyEvent e) {}
    @Override
    public void keyTyped(KeyEvent e) {}

    public static void main(String[] args) {
        JFrame frame = new JFrame("Snake Game");
SnakeGame game = new SnakeGame();
        frame.add(game);
frame.pack();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

4.OUTPUT


Snake Game
Game Over
SnakeGame.java
JPanel implements ActionListener, KeyListener {
IDTH = 600, HEIGHT = 400, BLOCK_SIZE = 20;
snake;
vEvent.VK_RIGHT;
false;
ERMINAL
PORTS
Data\Local\Programs\Eclipse Adoptium\jdk-17.0.8.101-hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMess
ages' '-cp' 'C:\Users\ASUS\AppData\Local\Temp\vscodesws_567de\jdt_ws\jdt.ls-java-project\bin' 'SnakeGame'
Debug: Sna...
Run: Snake...
Run: Snake...
CALL STACK
SnakeGame RUNNING
SnakeGame 2 RUNNING
BREAKPOINTS
Uncaught Exceptions
Caught Exceptions
Java: Ready
Spaces: 4
UTF-8
Java

5.Explanation of code

1. Class Definition:

```
public class SnakeGame extends JPanel implements ActionListener {
```

- SnakeGame class extends JPanel:
  - The game uses JPanel as its drawing surface to render the game components like the snake, food, and score.
- implements ActionListener:
  - The class implements ActionListener to handle the periodic updates for the game. This will allow us to control the movement of the snake at regular intervals using a timer.

2. Constants and Variables:

```
private static final int TILE_SIZE = 30;  // Size of each grid tile private

static final int WIDTH = 600;  // Width of the game window private

static final int HEIGHT = 600;  // Height of the game window private

static final int NUM_TILES_X = WIDTH / TILE_SIZE; private static

final int NUM_TILES_Y = HEIGHT / TILE_SIZE; private static final

int INITIAL_SNAKE_SIZE = 5;

private static final int GAME_SPEED = 100;  // Milliseconds for game refresh rate
```

- **TILE_SIZE**: The size of each individual grid square (30 pixels by 30 pixels).
- **WIDTH** and **HEIGHT**: These define the overall size of the game window in pixels (600x600 pixels).
- **NUM_TILES_X** and **NUM_TILES_Y**: These calculate how many grid tiles will fit horizontally and vertically based on the window size and TILE_SIZE.
- **INITIAL_SNAKE_SIZE**: The initial length of the snake when the game starts (5 segments).
- **GAME_SPEED**: The interval (in milliseconds) at which the game state will update. A value of 100 ms means the game updates every 100 milliseconds (or 10 times per second).

3. Instance Variables:

```
private LinkedList<Point> snake; private

Point food;

private int direction;  // 0 = up, 1 = right, 2 = down, 3 = left
```

```
private boolean gameOver;

private int score; private

Timer timer;
```

- **snake:** A LinkedList<Point> that stores the coordinates of each segment of the snake's body. Each Point object represents a grid tile where the snake is located.
- **food**: A Point representing the food's location in the grid.
- **direction**: An integer to represent the current direction the snake is moving:
  - 0 = up
  - 1 = right
  - 2 = down
  - 3 = left
- **gameOver:** A boolean that indicates whether the game is over. The game ends when the snake collides with the walls or itself.
- **score:** An integer that keeps track of the player's score, which increases each time the snake eats food.
- **timer:** A Timer object used to repeatedly trigger the actionPerformed() method to move the snake and refresh the screen.

4. Constructor:

```
public SnakeGame() {
    setPreferredSize(new Dimension(WIDTH, HEIGHT));
    setBackground(Color.BLACK);
    setFocusable(true);
```

- **setPreferredSize(new Dimension(WIDTH, HEIGHT)):** Sets the preferred size of the game window to 600x600 pixels.
- **setBackground(Color.BLACK):** Sets the background color of the game window to black.
- **setFocusable(true):** Makes the panel focusable, so it can capture keyboard events (like arrow key presses).

5. Initialize Game State:

```
snake = new LinkedList<>(); direction = 1;  // Initially moving right
score = 0; gameOver = false;
```

- Initializes the snake as an empty LinkedList.
- Sets the initial direction of the snake to "right" (direction = 1).
- Initializes the score to 0.
- Sets gameOver to false because the game hasn't ended yet.

6. Initialize Snake Body:

```
for (int i = 0; i < INITIAL_SNAKE_SIZE; i++) { snake.add(new Point(i, 0)); }
```

- This loop creates the initial snake body. The snake starts with INITIAL_SNAKE_SIZE segments.
- Each segment is represented by a Point object, where the x coordinate increases from left to right (starting from 0, 1, 2, etc.) and the y coordinate remains at 0 (meaning the snake starts at the top row of the grid).

7. Key Listener for Snake Movement:

```
addKeyListener(new KeyAdapter() {
   @Override public void keyPressed(KeyEvent e) { if (gameOver) return;

      switch (e.getKeyCode()) {
case KeyEvent.VK_UP:
            if (direction != 2) direction = 0;
break;
         case KeyEvent.VK_RIGHT:
            if (direction != 3) direction = 1;
break;
         case KeyEvent.VK_DOWN:
            if (direction != 0) direction = 2;
break;
         case KeyEvent.VK_LEFT:
```

```
            if (direction != 1) direction = 3;
break;
        }
    } });
```

- This code adds a KeyListener to capture user input from the keyboard.
- The keyPressed method listens for arrow key presses (UP, RIGHT, DOWN, LEFT).
- When a key is pressed, the direction of the snake is updated (unless the snake is already moving in the opposite direction). This ensures the snake cannot reverse direction directly (e.g., you can't go left if you're currently going right).

8. Game Timer: timer = new Timer(GAME_SPEED, this); timer.start();

- This creates a Timer object that triggers the actionPerformed method every GAME_SPEED milliseconds (100 ms, or 10 times per second).
- The Timer object is started immediately after creation.

9. Spawn Food:

```
spawnFood();
```

- This calls the spawnFood method to generate a random location for the food on the game grid.

10. Action Performed (Game Update Loop):

```
@Override public void
actionPerformed(ActionEvent e) {     if
(gameOver) return;
```

- This method is called every time the Timer ticks. It controls the movement of the snake and handles collisions.
- If the game is over (gameOver is true), the method returns early to stop any further game updates.

11. Move Snake:

```
Point head = snake.getFirst();
Point newHead = new Point(head);
```

```
switch (direction) {     case 0: newHead.y -
= 1; break;  // Up     case 1: newHead.x +=
1; break;  // Right     case 2: newHead.y +=
1; break;  // Down     case 3: newHead.x -=
1; break;  // Left
}
```

- **head:** The current position of the snake's head (first segment in the snake list).
- **newHead:** A copy of the current head.
- The switch statement updates the position of newHead based on the current direction:
  - If direction = 0 (up), subtract 1 from the y coordinate. ○ If direction = 1 (right), add 1 to the x coordinate. ○ If direction = 2 (down), add 1 to the y coordinate.
  - If direction = 3 (left), subtract 1 from the x coordinate.

12. Check for Collisions:

```
if (newHead.x < 0 || newHead.x >= NUM_TILES_X || newHead.y < 0 ||
    newHead.y >= NUM_TILES_Y || snake.contains(newHead)) {     gameOver = true;     repaint();
    return;
}
```

- **Wall Collision**: If newHead is outside the bounds of the grid (either x or y is less than 0 or exceeds the grid size), the game ends.
- **Self Collision:** If newHead is already part of the snake's body (snake.contains(newHead)), the game ends.
- If any collision occurs, the game is marked as over (gameOver = true), and the screen is updated with repaint().

13. Snake Eats Food:

```
if (newHead.equals(food)) {
score++;     spawnFood();
} else {     snake.removeLast();  // Remove tail if no
food eaten
}
```

- If the snake's head (newHead) is at the same location as the food, the snake eats the food:

    - The score is incremented.
    - A new food item is spawned at a random location.
- If no food is eaten, the snake's tail is removed (snake.removeLast()) to simulate the snake moving forward.

14. Add New Head:

```
snake.addFirst(newHead);
```

- The new head position is added to the front of the snake (addFirst()), making it the new head of the snake.

15. Repaint the Screen:

```
repaint();
```

CONCLUSION

The Snake game in Java is a fun and engaging project that showcases several important programming concepts, including object-oriented design, graphical user interface (GUI) development, and event handling. In this implementation, we created a basic version of the classic Snake game, which allows the player to control a snake that grows longer as it eats food, all while avoiding walls and its own tail. The game concludes when the snake collides with either the game board boundaries or its own body, at which point a "Game Over" message is displayed along with the player's score. In summary, this Snake game in Java offers a solid foundation for understanding basic game development principles. It can easily be expanded with new features and enhancements, making it a great starting point for beginners looking to explore game programming in Java.