Project Report

Name : Juturi Tejasri

Email id : juturitejasri@gmail.com

Internship : Java Development

Project Title : Snake Game in Java

Task Number : 01

Sponsor : Veritech

Language used : Java

Editor used : Java compiler editor

Task Description :

In this project,we aim to enhance the classic snake game from our childhood using advanced java concepts.The implementation involves a comprehensive understanding of object-oriented programming principles and the applications of Java Swing for creating an interactive graphical user interface.The snake will possess the ability to move in all four directions.the snake length will increase as it consumes food.The game concludes

when the snake either crossess itself or collides with perimeter of the game board.Food will be randomly placed at different positions within the game board.

- **STEPS TAKEN**

Create a Java Project

Begin by initiating a new Java project in Java compiler editor.

Creating a Java project:

Start by creating a new Java Project in Java compiler code editor.

Set up the game board:

Defining a grid or a game board where the snake and food will

move. You can represent the grid using a 2D array or any other suitable data structure.

```
public class SnakeGame {
    private static final int WIDTH = 20;
    private static final int HEIGHT = 20;
```

```
    private char[][] board = new char[WIDTH][HEIGHT];

    public SnakeGame() {
        // Initialize the board
        for (int i = 0; i < WIDTH; i++) {
            for (int j = 0; j < HEIGHT; j++) {
                board[i][j] = ' ';
            }
        }
    }
}
```

Create the Snake class:

Defining a class to represent the snake. The class should include properties such as length, direction, and body segments. Implement methods to move the snake, check for collisions, and grow the snake when it eats food.

```
public class Snake {
    private int length;
    private int[] x, y; // Arrays to store x and y coordinates of each body segment
    private char direction; // 'U' for up, 'D' for down, 'L' for left, 'R' for right

    public Snake(int initialLength) {
        length = initialLength;
        x = new int[length];
        y = new int[length];
        direction = 'R'; // Initially moving right
        // Initialize the snake's position
        for (int i = 0; i < length; i++) {
            x[i] = i;
            y[i] = 0;
        }
```

```
    }

    // Add methods to move the snake, check for collisions, and grow the snake
}
```

Creating the Food class:

Defining a class to represent the food. The food should appear at random positions on the game board.

```
public class Food {
    private int x, y;

    public Food() {
        // Generate random coordinates for the food
        x = (int) (Math.random() * WIDTH);
        y = (int) (Math.random() * HEIGHT);
    }

    // Add methods to get and set the food's position
```

```
}
```

Implement game logic:

Write the main game loop that controls the flow of the game. This loop should handle user input, update the positions of the snake and food, and check for collisions.

```
public class SnakeGame {
    // ...

    public void runGame() {
        // Implement the game loop
        // Handle user input, update snake and food positions, check for collisions
    }
}
```

Handle user input:

Allowing the player to control the direction of the snake using arrow keys or other input methods.

```
public class SnakeGame {
    // ...

    public void handleInput(char key) {
        // Update snake direction based on user input
    }
}
```

Displaying the game:

Using Java Swing or any other graphics library to create a graphicaluser interface for the game. Displaying the game board, snake, and food on the screen,and update their positions during each iteration of the game loop.

```
// Use JFrame, JPanel, and other Swing components to create the GUI
```

Handling collisions:

Checking for collisions between the snake, food, and game boundaries.End the game if the snake collides with itself or the boundaries, and update the score if the snake eats food.

```
// Implement collision detection logic
```

Adding scoring:

Keep track of the player's score based on the length of the snake and the number of food items eaten.

```
// Add scoring logic
```

Implement game over:

Displaying a game over message when the game ends, and allow the player to restart the game if desired.

```
// Implement game over logic
```

➢ CODE IMPLEMENTATION

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.util.ArrayList;
import java.util.Random;

public class SnakeGame extends JFrame implements ActionListener, KeyListener
{
    private static final int WIDTH = 20;
    private static final int HEIGHT = 20;
    private static final int TILE_SIZE = 20;
```

```
    private ArrayList<Point> snake;
    private Point food;
    private char direction;
    private boolean gameOver;
    private int score;

    public SnakeGame() {
        setTitle("Snake Game");
        setSize(WIDTH * TILE_SIZE, HEIGHT * TILE_SIZE);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        snake = new ArrayList<>();
        direction = 'R'; // Initially moving right

        // Initialize the snake
```

```
        snake.add(new Point(5, 5));
        snake.add(new Point(5, 4));
        snake.add(new Point(5, 3));

        spawnFood();

        // Set up timer for game loop
        Timer timer = new Timer(200, this);
        timer.start();

        addKeyListener(this);
        setFocusable(true);
        setVisible(true);
    }

    private void spawnFood() {
```

```
        Random rand = new Random();
        int x, y;
        do {
            x = rand.nextInt(WIDTH);
            y = rand.nextInt(HEIGHT);
        } while (snake.contains(new Point(x, y)));

        food = new Point(x, y);
    }

    private void moveSnake() {
        Point head = snake.get(0);
        Point newHead;

        switch (direction) {
            case 'U':
```

```
            newHead = new Point(head.x, (head.y - 1 + HEIGHT) % HEIGHT);
            break;
        case 'D':
            newHead = new Point(head.x, (head.y + 1) % HEIGHT);
            break;
        case 'L':
            newHead = new Point((head.x - 1 + WIDTH) % WIDTH, head.y);
            break;
        case 'R':
            newHead = new Point((head.x + 1) % WIDTH, head.y);
            break;
        default:
            throw new IllegalStateException("Unexpected value: " + direction);
    }

    if (snake.contains(newHead) || newHead.equals(food)) {
```

```
            if (newHead.equals(food)) {
                snake.add(0, food);
                score++;
                spawnFood();
            } else {
                gameOver = true;
            }
        } else {
            snake.add(0, newHead);
            snake.remove(snake.size() - 1);
        }
    }

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

```
            moveSnake();
            repaint();
        } else {
            JOptionPane.showMessageDialog(this, "Game Over!\nScore: " + score,
"Game Over", JOptionPane.INFORMATION_MESSAGE);
            resetGame();
        }
    }

    private void resetGame() {
        snake.clear();
        snake.add(new Point(5, 5));
        snake.add(new Point(5, 4));
        snake.add(new Point(5, 3));
        direction = 'R';
        spawnFood();
```

```
        gameOver = false;
        score = 0;
    }

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

        // Draw the snake
        g.setColor(Color.GREEN);
        for (Point p : snake) {
            g.fillRect(p.x * TILE_SIZE, p.y * TILE_SIZE, TILE_SIZE, TILE_SIZE);
        }

        // Draw the food
        g.setColor(Color.RED);
```

```
        g.fillRect(food.x * TILE_SIZE, food.y * TILE_SIZE, TILE_SIZE, TILE_SIZE);

        // Draw the score
        g.setColor(Color.BLACK);
        g.drawString("Score: " + score, 10, 20);

        // Draw the game board grid
        g.setColor(Color.GRAY);
        for (int i = 0; i < WIDTH; i++) {
            g.drawLine(i * TILE_SIZE, 0, i * TILE_SIZE, HEIGHT * TILE_SIZE);
        }
        for (int i = 0; i < HEIGHT; i++) {
            g.drawLine(0, i * TILE_SIZE, WIDTH * TILE_SIZE, i * TILE_SIZE);
        }
    }
```

```
@Override
public void keyPressed(KeyEvent e) {
    int keyCode = e.getKeyCode();

    switch (keyCode) {
        case KeyEvent.VK_UP:
            if (direction != 'D') {
                direction = 'U';
            }
            break;
        case KeyEvent.VK_DOWN:
            if (direction != 'U') {
                direction = 'D';
            }
            break;
        case KeyEvent.VK_LEFT:
```

```
            if (direction != 'R') {
                direction = 'L';
            }
            break;
        case KeyEvent.VK_RIGHT:
            if (direction != 'L') {
                direction = 'R';
            }
            break;
    }
}

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

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

    public static void main(String[] args) {
        new SnakeGame();
    }
}
```

- **CHALLENGES FACED:**

Responsiveness:

The game responsiveness might be improved by handling key events more efficiently. Currently, key events are processed in the keyPressed method, and changes in the direction are checked directly. Consider using a separate thread or a more responsive event handling mechanism.

Magic Numbers:

The use of magic numbers, such as 200 in the Timer constructor, should be avoided. Consider defining constants or variables with meaningful names to enhance code readability and maintainability.

Code Duplication:

There is a certain degree of code duplication, especially in the initialization of the snake in the constructor and the resetGame method. Extracting common logic into a separate method can enhance code maintainability.

Drawing Logic:

The drawing logic in the paint method could be encapsulated into separate methods for drawing the snake, food, score, and game board grid. This would improve code readability and maintainability.

Exception Handling:

The IllegalStateException thrown in the moveSnake method could be replaced with a more specific exception or handled differently to provide more information about the unexpected value.

Game Over Message:

The JOptionPane used for displaying the game over message is a blocking call, pausing the game. Consider using a non-blocking approach to allow the player to restart the game without closing the application.

Scalability:

The game board size is fixed at WIDTH = 20 and HEIGHT = 20. Consider making these parameters configurable to allow for variations in the game board size.

Documentation:

While the code is generally well-commented, additional comments could be added to explain complex logic, especially in the moveSnake method, to aid understanding.

Encapsulation:

The SnakeGame class could benefit from encapsulating certain functionalities into separate classes or methods. For instance, logic related to drawing, game state management, and user input handling could be separated.

Game Difficulty:

Adding different difficulty levels or speed adjustments could enhance the gameplay experience. Currently, the snake moves at a fixed speed, and increasing difficulty levels could make the game more challenging.

- **LEARNINGS:**

Java Swing for GUI:

The program uses Java Swing components (JFrame, Graphics, JOptionPane) to create a graphical user interface for the Snake game. The game window, snake, food, and score are displayed using Swing components.

Event Handling:

The program implements the ActionListener and KeyListener interfaces to handle events. The actionPerformed method is invoked by a Timer to control the game loop, and the keyPressed method is used to handle user input for changing the direction of the snake.

Game Loop and Timer:

The game operates on a continuous loop controlled by a Timer. The actionPerformed method is called at regular intervals, allowing the game to update the snake's position, check for collisions, and repaint the screen.

Snake Movement and Collision Detection:

The moveSnake method is responsible for updating the snake's position based on the current direction. It checks for collisions with the game boundaries, the snake itself, and the food. If the snake eats food, its length increases, and a new food location is generated.

Game Over and Reset:

The game ends if the snake collides with itself or the boundaries. A game over message is displayed using JOptionPane, showing the final score. The resetGame method is called to reset the game when the player decides to play again.

Random Food Placement:

The spawnFood method generates a random position for the food, ensuring that it does not overlap with the snake's body.

Drawing on Graphics:

The paint method is overridden to draw the snake, food, score, and the game board grid on the screen. Different colors are used for the snake, food, score, and grid.

Score Tracking:

The program tracks the player's score, which increases each time the snake eats food. The score is displayed on the game screen.

User Input Handling: Arrow key events (keyPressed method) are used to change the direction of the snake. The snake cannot reverse its direction completely to avoid self-collisions.

Code Organization:

The code is organized into methods for better readability and maintainability. The main logic is in the actionPerformed, moveSnake, and other methods.

CONCLUSION:

In conclusion, the Snake Game project serves as a practical and enjoyable exercise for learning and applying Java programming skills, making it a valuable project for those exploring game development and GUI programming in Java.