Java Snake Game: Project Report

Team Snake Game

August 5, 2025

Contents

1 Project Overview

Our team of three developed a classic Snake Game using Java Swing as our academic project to demonstrate proficiency in object-oriented programming (OOP), GUI development, and event handling. The game involves a snake that moves across a grid, eats apples to grow, and avoids collisions with itself and the boundaries. The project showcases our ability to plan and implement a cohesive application using modular design, user input handling, and graphical rendering. This report outlines the modules used, header files (Java imports), OOP principles applied, keyboard input mechanisms, GUI implementation, and how these components integrate to create a functional game.

2 Modules Used

The project is organized into four Java files, each serving a specific purpose:

- `SnakeGame.java`: The main class that initializes the game window and launches the application.
- `GamePanel.java`: The core game panel that manages the game loop, rendering, and user input.
- `Snake.java`: Represents the snake, handling its movement, growth, and direction.
- `Apple.java`: Manages the apple's position and spawning logic.

These modules work together to create a modular and maintainable codebase, with each class encapsulating specific functionality.

3 Header Files (Java Imports)

The following Java packages and classes are imported to support the game's functionality:

- `javax.swing.*` (`SnakeGame.java`, `GamePanel.java`): Provides GUI components like `JFrame`, `JPanel`, and `JButton` for the game window and interface.
- `java.awt.*` (`GamePanel.java`): Supplies graphics tools (`Graphics`, `Color`, `Font`) and event handling (`ActionListener`, `KeyAdapter`).
- `java.awt.event.*` (`GamePanel.java`): Enables keyboard and action event handling (`ActionEvent`, `KeyEvent`).
- `java.util.Random` (`Apple.java`): Used for generating random positions for the apple.

These imports provide the necessary tools for GUI rendering, event handling, and random number generation, ensuring the game operates smoothly.

4 Object-Oriented Programming (OOP) Implementation

We applied several OOP principles to ensure a robust and scalable design:

- Encapsulation:
  - In `Snake.java`, the snake's state (position arrays `x`, `y`, `bodyParts`, `direction`) is encapsulated as instance variables, with methods like `move()`, `grow()`, and `setDirection()` controlling access and updates.
  - In `Apple.java`, the apple's coordinates (`x`, `y`) are encapsulated, with `spawnApple()` managing their updates.
- Abstraction:
  - The `GamePanel` class abstracts the game loop and rendering logic, exposing only high-level methods like `startGame()` and `restartGame()`.
  - The `Snake` and `Apple` classes abstract their respective behaviors, allowing `GamePanel` to interact with them without needing to know their internal implementations.
- Modularity:
  - Each class has a single responsibility: `SnakeGame` for window setup, `GamePanel` for game logic, `Snake` for snake behavior, and `Apple` for apple spawning.
  - This separation allows easy maintenance and potential extensions, such as adding new features or modifying existing ones.
- Inheritance and Polymorphism:
  - `GamePanel` extends `JPanel` and implements `ActionListener`, inheriting Swing's panel functionality and overriding methods like `paintComponent()` and `actionPerformed()`.
  - The inner class `MyKeyAdapter` extends `KeyAdapter`, overriding `keyPressed()` to handle keyboard input.

These principles ensure the codebase is organized, reusable, and easy to understand.

5 Keyboard Input Handling

Keyboard input is handled to control the snake's direction using both arrow keys and W/A/S/D keys:

- Implementation: The `GamePanel` class includes an inner class `MyKeyAdapter` that extends `KeyAdapter`. The `keyPressed(KeyEvent e)` method captures key events and maps them to snake directions:
  - Left arrow or A: Sets direction to 'L'.
  - Right arrow or D: Sets direction to 'R'.
  - Up arrow or W: Sets direction to 'U'.
  - Down arrow or S: Sets direction to 'D'.

- Logic: The `setDirection(char newDir)` method in `Snake` ensures the snake cannot reverse directly (e.g., from left to right), preventing invalid moves.
- Integration: The `GamePanel` constructor adds the key listener (`this.addKeyListener(new MyKeyAdapter())`). The panel is set to be focusable (`this.setFocusable(true)`) to receive keyboard events. After a game over, `restartGame()` calls `requestFocusInWindow()` to ensure input is captured.

This approach provides a responsive and intuitive control scheme for players.

6 GUI Implementation

The game's graphical user interface is built using Java Swing:

- Window Setup (`SnakeGame.java`):
  - Creates a `JFrame` titled "Snake Game".
  - Adds a `GamePanel`, sets it as non-resizable, and centers it using `setLocationRelativeTo(nul`
  - Uses `pack()` to fit the frame to the panel's preferred size (600x600 pixels).
- Game Rendering (`GamePanel.java`):
  - Overrides `paintComponent(Graphics g)` to call `draw(Graphics g)`, which renders game elements.
  - The `draw` method:
    - Draws the apple as a red oval using `fillOval()`.
    - Draws the snake, with the head in green and body in a darker green using `fillRect()`.
    - Displays the score using `drawString()`.
  - The `gameOver` method displays "Game Over" and the final score, showing a "Play Again" button.
- Interactive Elements:
  - A `JButton` ("Play Again") is added to the panel, centered at game over, and triggers `restartGame()` when clicked.
  - The panel uses a black background and a grid-based rendering system (25x25 pixel units).

The GUI provides a clear and engaging visual experience, with real-time updates driven by a `Timer`.

7 How It All Works Together

The components integrate seamlessly to create a functional game:

- Initialization: `SnakeGame` creates the `JFrame` and adds `GamePanel`, which initializes the game state (`startGame()`) with a new `Snake` and `Apple`.
- Game Loop: `GamePanel` uses a `Timer` (100ms delay) to trigger `actionPerformed()`, which:
  - Calls `Snake.move()` to update the snake's position.
  - Invokes `checkApple()` to detect apple collisions, triggering `Snake.grow()` and `Apple.spawnApple()` if needed.
  - Calls `checkCollisions()` to detect game-over conditions.
  - Triggers `repaint()` to update the GUI.
- User Input: Keyboard events are captured by `MyKeyAdapter`, updating the snake's direction via `Snake.setDirection()`.
- Rendering: The `draw` method renders the snake, apple, and score, or the game-over screen if the game ends.
- Restart Mechanism: The ”Play Again” button calls `restartGame()`, resetting the game state and restarting the timer.

This modular design ensures each component handles a specific aspect of the game, with clear interactions between classes. The use of OOP principles, event-driven programming, and Swing's GUI capabilities results in a cohesive and playable game.