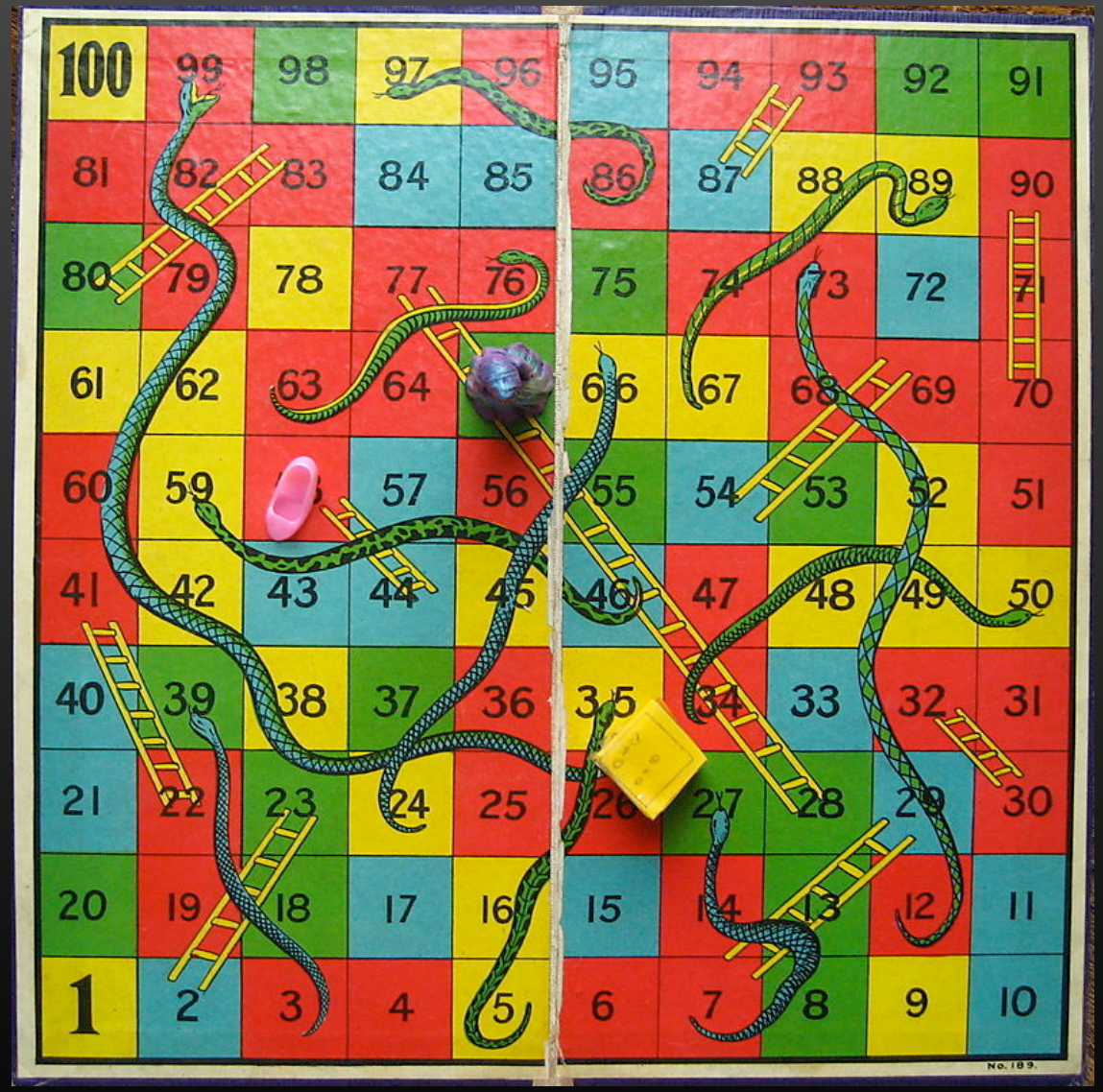
Snakes and Ladders



By
Niraj(1BM24CS191)
Niraj V(1BM24CS192)
Prajwal Vasantha Kumar(1BM24CS210)
Pranav Hebbar K(1BM24CS214)

Project Overview

- This mini-project involved the digital re-creation of the classic board game, Snakes and Ladders, using Java. The primary objective was to develop an interactive and engaging game where players navigate a board based on dice rolls, striving to reach the final square while encountering strategic snakes and helpful ladders.
- Classic board game recreated digitally using Java
- Objective: Navigate players from start to finish using dice rolls
- Incorporates snakes (slide down) and ladders (climb up) mechanics
- Supports multiplayer gameplay with engaging UI

Problem Statement

Develop a modular version of 'Snakes and Ladders' using Object-Oriented Programming (OOP) principles in Java. The project aims to solve the problem of hard-coded game logic by creating distinct classes for Player, Board, Entity (Snakes/Ladders), and Game Engine. The system should support a dynamic number of players and allow for easily configurable board layouts, demonstrating effect

System Architecture

- Player Class : Tracks player ID, current position on the board, and final rank in the game.
- Board Class : Manages the layout of snakes and ladders, and controls valid player movements.
- Dice Class : Simulates random dice rolls (1-6) and manages rules for extra turns on specific rolls.
- Game Class : Orchestrates overall game flow, player turns, and determines victory conditions.

OOP’s Concepts Used

A. Encapsulation

Encapsulation is the practice of bundling data (fields) and methods together within a class while restricting direct access to the internal data.

Implementation in the Project:

- Player Class: Fields such as name, id, currentPosition, and color are declared as private. Access to these fields is controlled through public getter and setter methods (e.g., getPosition() and setPosition()), ensuring data integrity.
- Board Class: Internal data structures such as the maps for snakes and ladders are kept private. Other classes interact with this data through controlled methods like getSnakeTail(int head), preventing unintended modifications.

B. Inheritance

Inheritance allows a class to acquire properties and behaviors from another class, promoting code reuse.

Implementation in the Project:

- GameApp extends Application: The main application class inherits from the JavaFX Application class, gaining access to lifecycle methods required to launch and manage a windowed application.
- Dice extends VBox: The Dice class inherits from VBox, making it a self-contained UI component. This allows the dice to manage both its visual layout and rolling logic while being easily added to the scene graph.

OOP’s Concepts Used

C. Abstraction

Abstraction hides complex implementation details and exposes only essential features to the user.

Implementation in the Project:

- Interface-Based Collections: Collections are declared using interfaces such as List and Map rather than concrete implementations like ArrayList or HashMap. This allows flexibility and loose coupling.
- Functional Abstraction in the Board Class: The Board class abstracts the complexity of drawing the grid, snakes, and ladders. The GameController simply calls methods like initializeSnakesAndLadders() without needing to understand the underlying mathematical calculations or drawing logic.

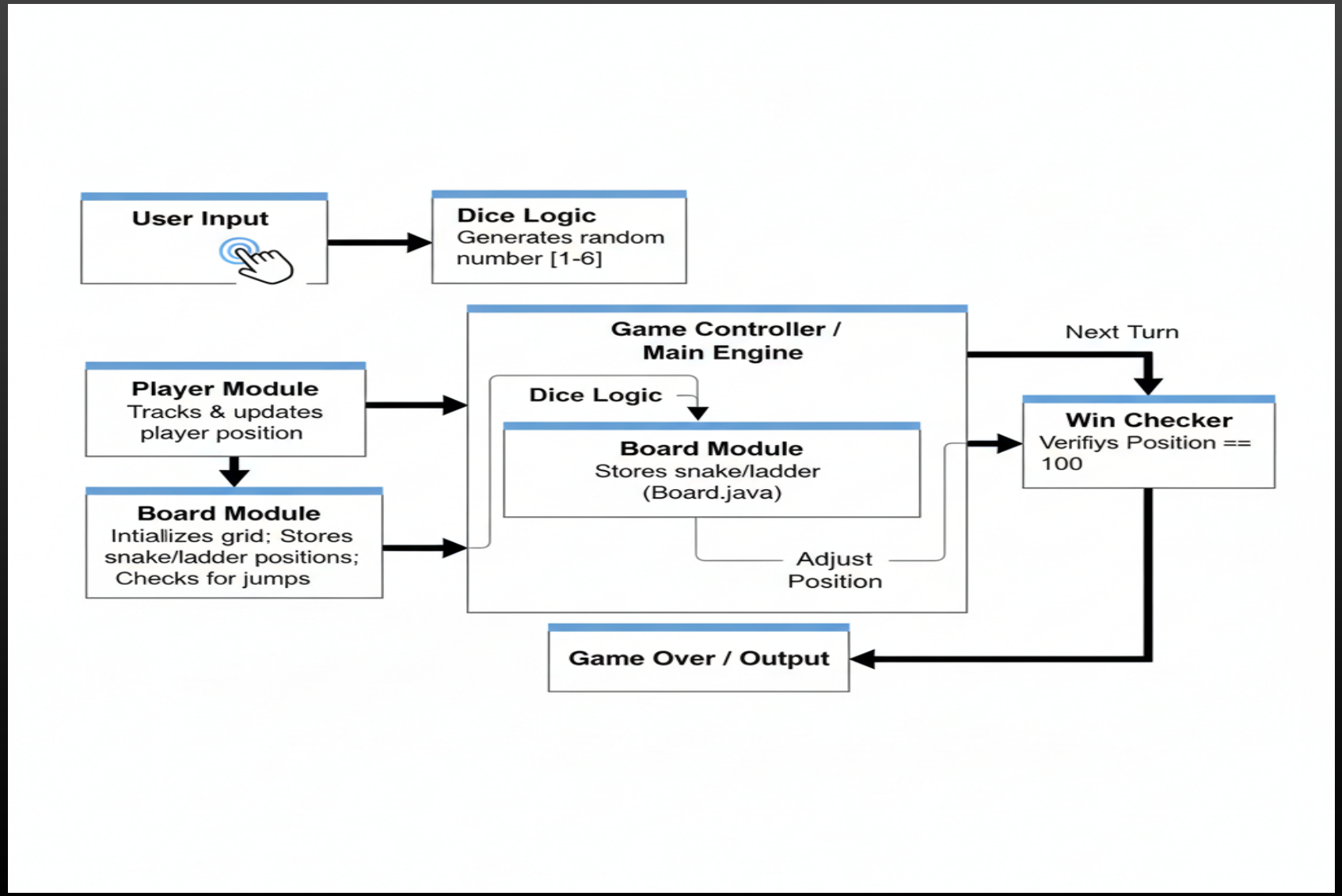
D. Polymorphism

Polymorphism allows methods or objects to behave differently based on their implementation or context.

Implementation in the Project:

- Method Overriding: The start(Stage primaryStage) method in GameApp overrides the method from the Application class. When Application.launch() is called, JavaFX polymorphically executes the overridden start method.
- Event Handling with Lambdas: Lambda expressions such as rollButton.setOnAction(e -> { ... }) treat blocks of behavior as objects, demonstrating functional polymorphism through event-driven programming.

Modules


User Input
Dice Logic
Generates random number [1-6]
Game Controller / Main Engine
Next Turn
Player Module
Tracks & updates player position
Dice Logic
Board Module
Stores snake/ladder (Board.java)
Win Checker
Verifiys Position == 100
Board Module
Intiallizes grid; Stores snake/ladder positions; Checks for jumps
Adjust Position
Game Over / Output

Modules

1. GameApp.javaPurpose:

This is the entry point of the application.

Key Explanation:It extends the JavaFX Application class, which is required to start any JavaFX program.The start() method is the first method called by the system. It initializes the GameController and sets up the primary window (Stage).The main() method simply calls launch(), kicking off the JavaFX lifecycle.

2. GameController.javaPurpose:

The "Brain" of the application. It manages the game flow and connects the logic to the user interface.

Key Explanation:Game Loop: It manages whose turn it is (currentPlayerIndex), checks for win conditions, and handles the flow of the game.Event Handling: It listens for the Dice roll events. When the dice is rolled, it calculates the new position (movePlayer).

Rule Enforcement: It checks if a player lands on a Snake or Ladder (checkTileEvents) and moves them accordingly.

UI Management: It creates the side panel (title, labels) and handles the "Game Setup" dialog where users choose the number of players.

Modules

3. Board.java

Purpose: Handles the Visual Representation and Physical Layout of the game board.Key Explanation:

Grid System: It mathematically generates the 10x10 grid. It calculates the exact X and Y coordinates for every number (1-100) using the getCoordinatesForNumber() method, handling the zig-zag pattern (left-to-right, then right-to-left).

Drawing Graphics: It actually draws the shapes. It uses JavaFX Line and QuadCurve to draw the green Ladders and red Snakes dynamically based on coordinates.

Data Structure: It stores the locations of snakes and ladders in a Map (e.g., Key: 98 -> Value: 78 means a snake at 98 goes down to 78).

4. Player.java

Purpose: Represents a Participant in the game. It is a Model class holding data.Key

Explanation:

State Management: It keeps track of the player's unique details: Name, ID, Color, and —most importantly—their currentPosition (1-100).Visual Token: It "owns" a JavaFX Circle object representing the player on the screen.

Animation: It contains the animateMove() method, which creates a smooth sliding animation (TranslateTransition) when the player moves from one spot to another.

Modules

5. Dice.java

Purpose: A custom UI Component that behaves like a physical die.Key Explanation:

Custom Component: It extends VBox, meaning it's a visual box that can be added directly to the layout.

Random Logic: It uses the java.util.Random class to generate a number between 1 and 6.Animation: It performs a 360-degree spin animation (RotateTransition) to simulate rolling before showing the final number.

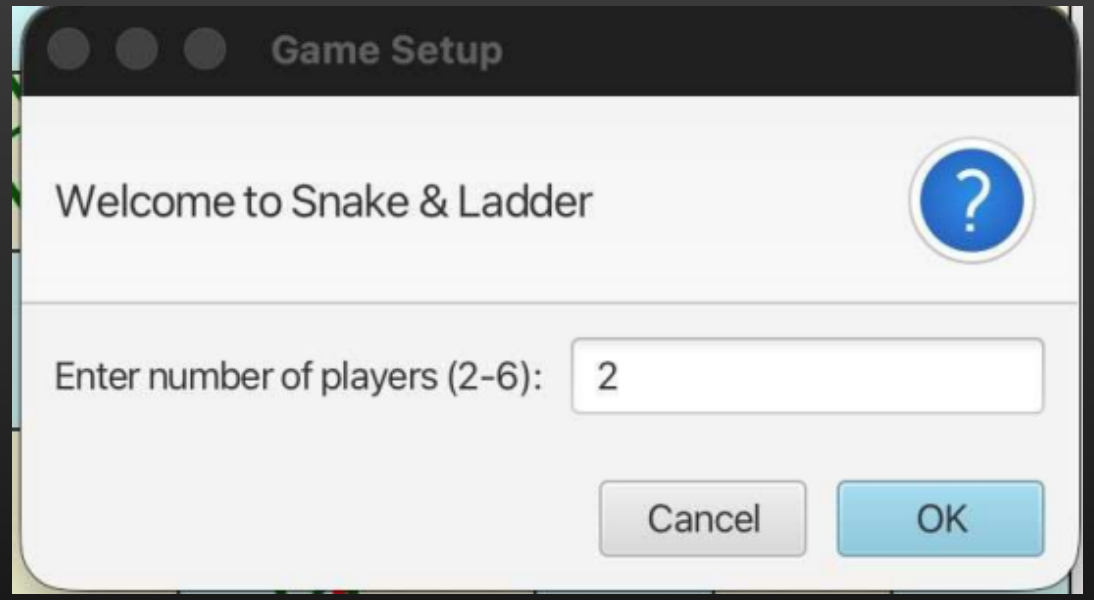
6. Tile.java

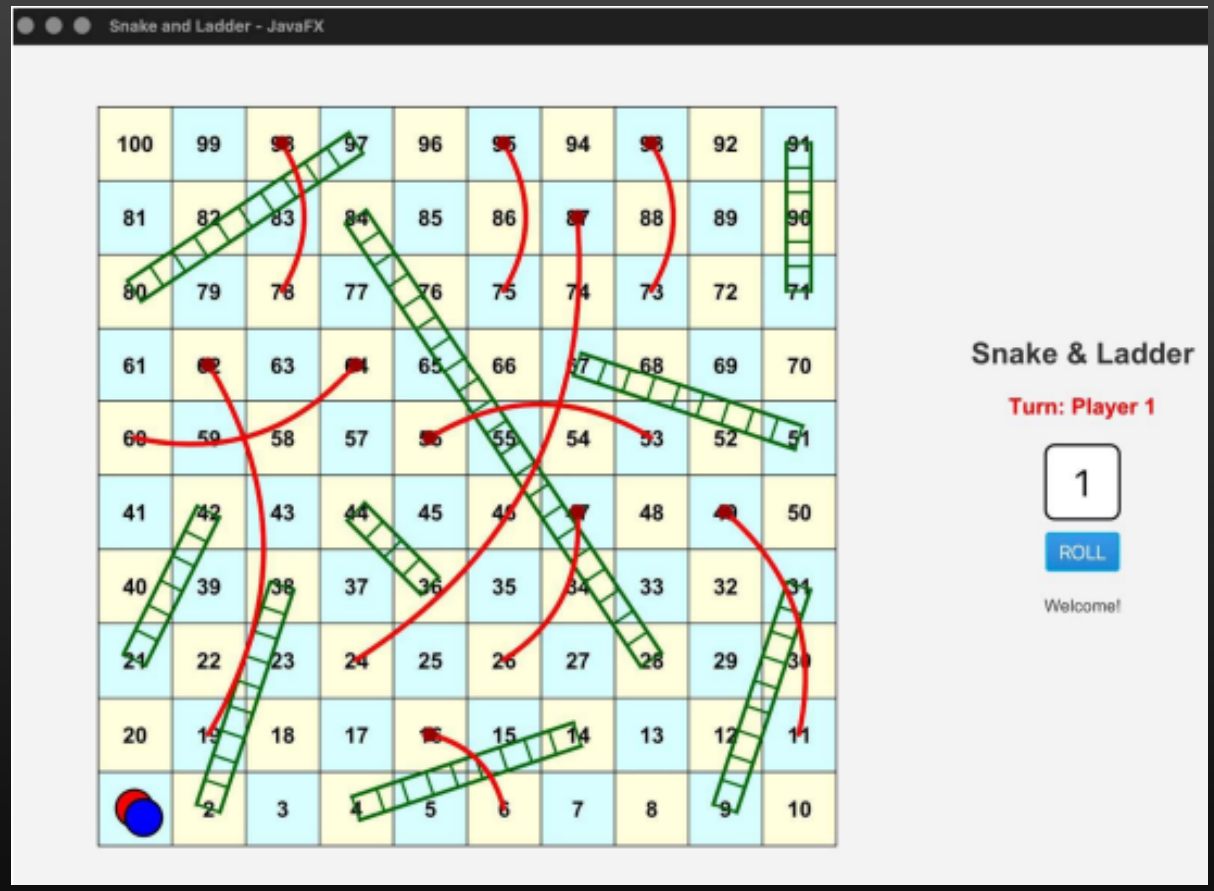
Purpose: Represents a Single Square on the grid.

Key Explanation: It extends StackPane, allowing it to stack a square shape and a text number on top of each other.

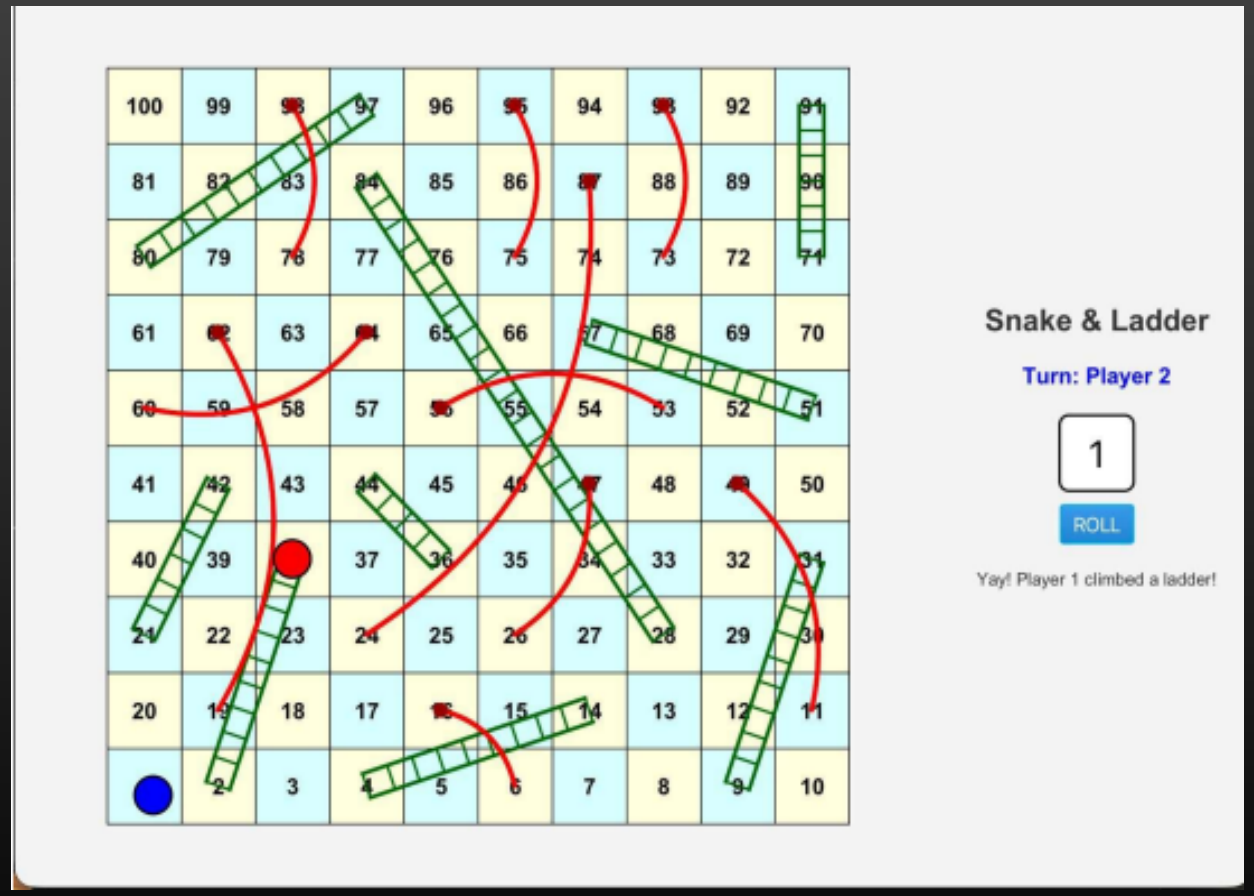
It determines its own color based on whether the number is even or odd (Checkerboard pattern), making the board easier to read.

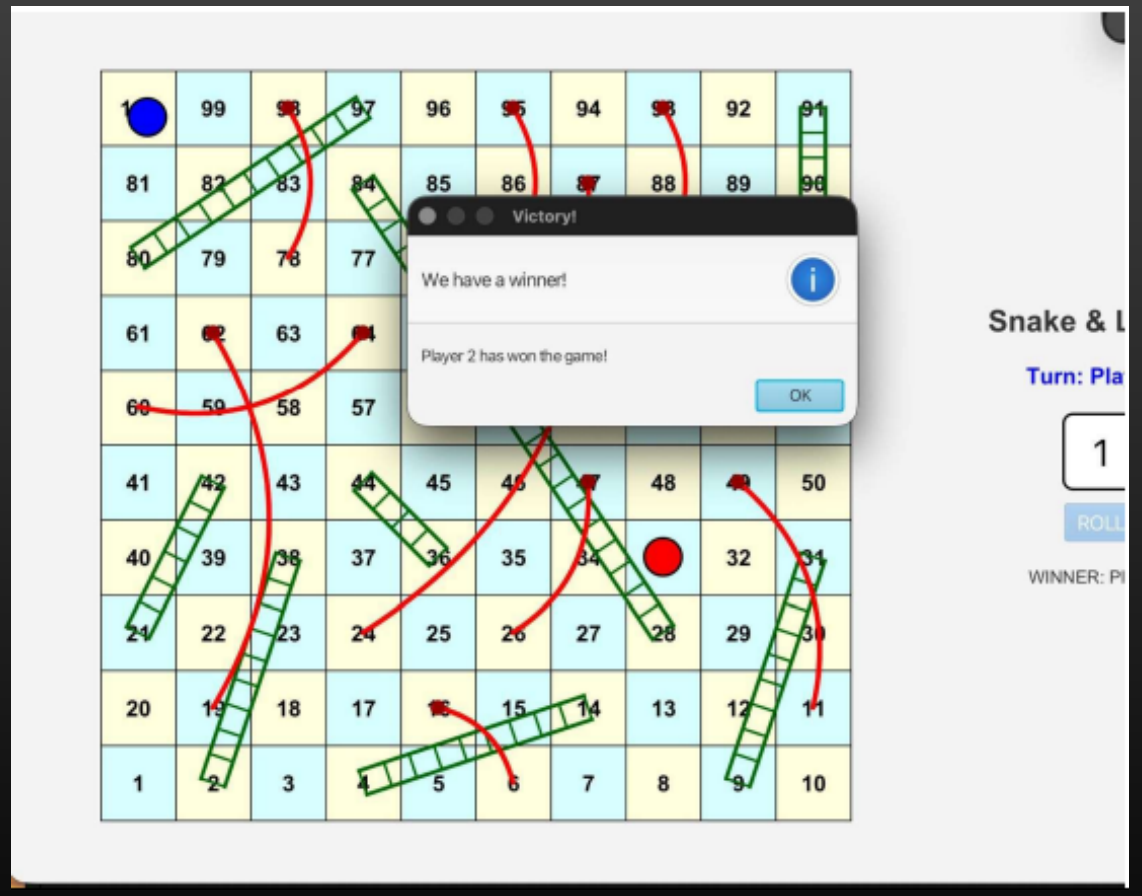
Results


Game Setup
Welcome to Snake & Ladder
Enter number of players (2-6):
2
Cancel
OK


Snake and Ladder - JavaFX
Snake & Ladder
Turn: Player 1
1
ROLL
Welcome!

Results


Snake & Ladder
Turn: Player 2
1
ROLL
Yay! Player 1 climbed a ladder!


Victory!
We have a winner!
Player 2 has won the game!
OK
Snake & L
Turn: Pla
1
ROLL
WINNER: Pl

Conclusion

In summary, this project successfully translates the entire logic of Snakes and Ladders into a structured Java application. By developing individual modules for the Board, Players, and Dice, I have created a system where separate components work in harmony to simulate a complex, turn-based environment.

Thank You