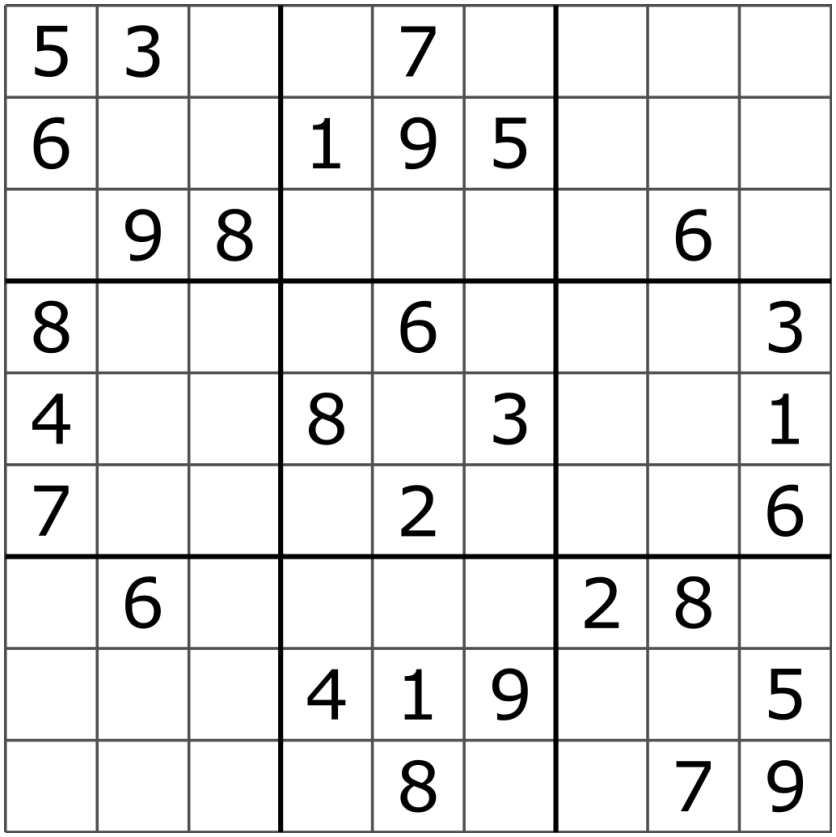
Sudoku Checker

Made by Daphne Baui and Mark Olaguera

CMSC 125

How the Program Works

What is Sudoku?



An example of a Sudoku Puzzle

Sudoku is a game of Japanese origin wherein the main objective is to fill up all the squares in a 9 by 9 square grid using numbers. Like all games, there are rules in playing Sudoku. One such rule is that in the grid, all columns and all rows must have no duplicate numbers (i.e. having two nines in a column or two sixes in a row). Another rule in Sudoku is that all 9 sub-tables must have all numbers from 1 to 9.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

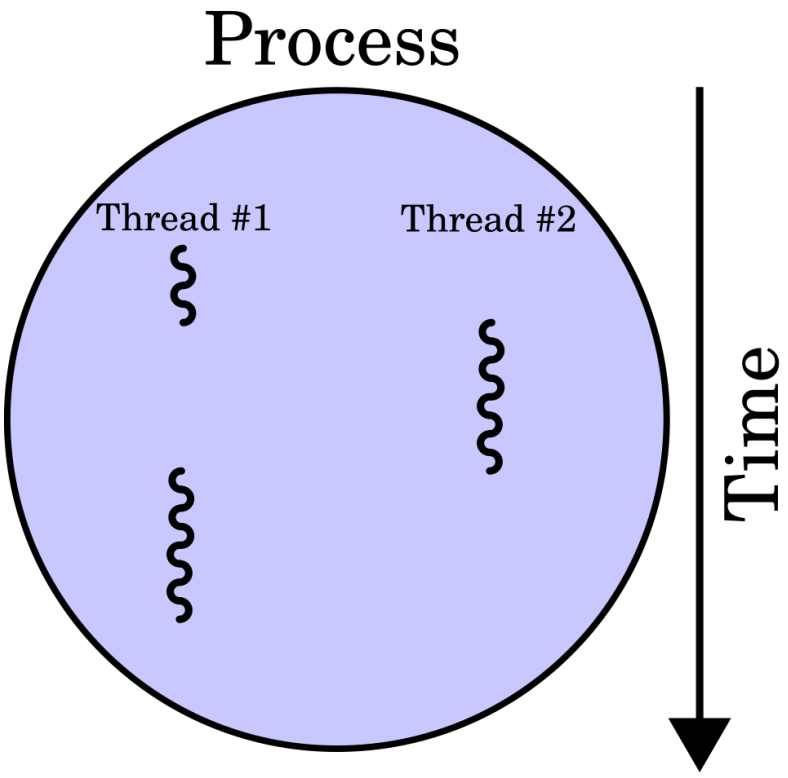
An example of a solved Sudoku Puzzle, the red digits are those filled in to complete the puzzle

A fun fact about Sudoku is that the fewest number of clues that is needed to properly solve a puzzle is 17 digits. Another fun fact about the game is that Sudoku in its most popular iteration only appeared in the 1980s, even though puzzles similar to Sudoku appeared as early as the 19th century.

How the Program Solves Sudoku Problems

There are many ways Sudoku can be solved, one such case is through the use of multithreading. In Operating Systems, multithreading is the ability of a Central Processing Unit (the computer's "brain") to provide multiple threads of execution. In layman's terms, this is the ability of a computer to execute multiple "threads", which are the smallest sequence of programmed instructions that can be executed, almost simultaneously through the use of a scheduler; which is responsible for allotting which threads should be executed. These threads that are being executed are part of what is

called a process, which is formed as a result of combining multiple threads.



An illustration of multithreading, two threads are being executed in a process

What the Program Contains

In this program, we have five classes contained under two packages, namely the “logic” and “gui” packages. These classes have their own unique tasks to do, and some of these contain the concept of multithreading.

Under the “logic” package, it contains the backdoor logic that would make the entire program functional in the first place; it contains the “PuzzleGenerator” and “SudokuValidator” classes. Under the “gui” package, it contains the classes that would constitute as the User Interface of the program. It contains the “StartPopup”, “SudokuGame”, and “SudokuGUI” classes.

PuzzleGenerator Class

The PuzzleGenerator class is responsible for generating the Sudoku puzzle. This class, alongside the SudokuValidator class, is known as the model class, which is responsible for setting up the internal logic of the program.

It has seven methods that all ensure that the puzzle exists, from generating the puzzle and checking that there are no redundancies to determining how many and where the digits should be placed in the puzzle.

Within the PuzzleGenerator class, there are attributes responsible for generating the puzzle itself. In this case, the number of pre-filled cells decreases by difficulty.

```Java
package logic;

import java.util.Random;

public class PuzzleGenerator {

    private static final int SIZE = 9;
    private static final int[] EASY = {40, 45, 50};   // Number of pre-filled
cells for different difficulties
    private static final int[] MEDIUM = {30, 35, 39};
    private static final int[] HARD = {20, 25, 29};

    private int[][] board;

    public PuzzleGenerator() {
        board = new int[SIZE][SIZE]; // Initialize a 9x9 board
        // Fill the board with zeros initially
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                board[i][j] = 0;
            }
        }
    }
```

The generatePuzzle method, like the aforementioned method, also can generate the puzzle; however, it generates a fully-solved board. To counteract this issue, this method also removes numbers on random positions to ensure that the player may input their solutions.

```java
public int[][] generatePuzzle(String difficulty) {
    int numClues = getCluesForDifficulty(difficulty);

    // Generate a full solved Sudoku board first
    generateSolvedBoard();

    // Randomly remove numbers based on the difficulty
    removeClues(numClues);

    return board;
}
```

The generateSolvedBoard method backtracks to generate a solved Sudoku board. This calls on the generatePuzzle method.

```java
private void generateSolvedBoard() {
    // Simple backtracking to generate a solved Sudoku board
    solve(board);
}
```

The solve method is responsible for solving the Sudoku grid itself. It does it through the use of a basic backtracking algorithm. This may be considered a 2D array since it has a 9 by 9 grid, and each element corresponds to a number in the board.

```java
private boolean solve(int[][] board) {
    // Basic backtracking algorithm to solve the Sudoku grid
    for (int row = 0; row < SIZE; row++) {
        for (int col = 0; col < SIZE; col++) {
            if (board[row][col] == 0) { // Find empty cell
```

```
                    for (int num = 1; num <= SIZE; num++) {
                        if (isValid(board, row, col, num)) {
                            board[row][col] = num;
                            if (solve(board)) {
                                return true;
                            }
                            board[row][col] = 0;
                        }
                    }
                    return false; // No valid number found
                }
            }
        }
        return true;
    }
```

The isValid method checks if the player inputted all the empty squares in the puzzle. This is also a 2d array

Java
```
    private boolean isValid(int[][] board, int row, int col, int num) {
        for (int i = 0; i < SIZE; i++) {
            if (board[row][i] == num || board[i][col] == num || board[row - row
% 3 + i / 3][col - col % 3 + i % 3] == num) {
                return false;
            }
        }
        return true;
    }
```

The removeClues method is responsible for removing any unnecessary clues that would've helped the player in filling up the squares, which would've defeated the purpose of the game.

Java
```
    private void removeClues(int numClues) {
        Random random = new Random();
        int cluesRemoved = 0;
```

```
        while (cluesRemoved < (SIZE * SIZE - numClues)) {
            int row = random.nextInt(SIZE);
            int col = random.nextInt(SIZE);

            if (board[row][col] != 0) {
                board[row][col] = 0;
                cluesRemoved++;
            }
        }
    }
```

Lastly, the getCluesForDifficulty is the method that determines how many clues a puzzle should have, depending on the difficulty the player chose. In switching difficulties, if the difficulty chosen is invalid, then it would default to Easy mode.

```Java
    private int getCluesForDifficulty(String difficulty) {
        switch (difficulty) {
            case "Easy":
                return EASY[new Random().nextInt(EASY.length)];
            case "Medium":
                return MEDIUM[new Random().nextInt(MEDIUM.length)];
            case "Hard":
                return HARD[new Random().nextInt(HARD.length)];
            default:
                return 36; // Default to Easy if invalid difficulty
        }
    }
}
```

SudokuValidator Class

The SudokuValidator class, on the other hand, has four methods that check if the user input for the puzzle is correct. This class is where the concept of threads and concurrency are implemented.

Within the SudokuValidator class, three interfaces are included, and all of them include the “concurrent” prefix, which is used to enable concurrency in a program, which would be important for enabling threads to work. The interface “ExecutorService” is responsible for terminating and tracking progress of asynchronous tasks, “Executor” is responsible for executing runnable tasks, and “AtomicBoolean” is responsible for allowing booleans to be updated atomically, that is, to allow individual threads to check their boolean values.

Aside from that,there is a method called “validate”, and it includes “newFixedThreadPool”, which creates a thread pool. A thread pool is a collection of threads that are maintained to be allocated for concurrent execution. This avoids any potential deadlocks. Within the validate method as well, there are loops that will check if the user input is correct for both the rows and columns, it calls on the other methods to work.

The executor.submit method is used to execute a task some other time in the future, which in this case, would be used to return a future value (this is called a future, or an object that stores values from the future, in this case, the results of whether or not the threads are valid).

The reason why threads are used in this instance is because they are tied already to the GUI, and without them, the program would not function properly.

```Java
package logic;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicBoolean;

public class SudokuValidator {
    private static final int SIZE = 9;

    public boolean validate(int[][] board) {
```

```
        ExecutorService executor = Executors.newFixedThreadPool(SIZE * 3); //
For rows, columns, and sub-grids
        AtomicBoolean isValid = new AtomicBoolean(true);

        // Validate rows
        for (int row = 0; row < SIZE; row++) {
            int finalRow = row;
            executor.submit(() -> {
                if (!isRowValid(board, finalRow)) {
                    isValid.set(false);
                }
            });
        }

        // Validate columns
        for (int col = 0; col < SIZE; col++) {
            int finalCol = col;
            executor.submit(() -> {
                if (!isColumnValid(board, finalCol)) {
                    isValid.set(false);
                }
            });
        }

        // Validate sub-grids
        for (int grid = 0; grid < SIZE; grid++) {
            int finalGrid = grid;
            executor.submit(() -> {
                if (!isGridValid(board, finalGrid)) {
                    isValid.set(false);
                }
            });
        }

        executor.shutdown();

        // Wait for all threads to complete
        while (!executor.isTerminated()) {
            // Loop until all tasks are done
        }

        return isValid.get();
    }
```

For this method called isRowValid, it sets the conditions if each row that the player inputted is a valid row, meaning that there are no duplicates.

```java
private boolean isRowValid(int[][] board, int row) {
    boolean[] seen = new boolean[SIZE + 1]; // 1 to 9
    for (int col = 0; col < SIZE; col++) {
        int value = board[row][col];
        if (value < 1 || value > SIZE || seen[value]) {
            return false;
        }
        seen[value] = true;
    }
    return true;
}
```

For this method called isColumnValid, on the other hand; it sets the conditions if the columns are valid columns, which means if this too has no duplicate values.

```java
private boolean isColumnValid(int[][] board, int col) {
    boolean[] seen = new boolean[SIZE + 1];
    for (int row = 0; row < SIZE; row++) {
        int value = board[row][col];
        if (value < 1 || value > SIZE || seen[value]) {
            return false;
        }
        seen[value] = true;
    }
    return true;
}
```

Lastly, the method called isGridValid checks on the entire grid itself if the values inputted are valid. In a sense, it is an extension of both isRowValid and isColumnValid.

```java
    private boolean isGridValid(int[][] board, int grid) {
        boolean[] seen = new boolean[SIZE + 1];
        int startRow = (grid / 3) * 3;
        int startCol = (grid % 3) * 3;

        for (int row = startRow; row < startRow + 3; row++) {
            for (int col = startCol; col < startCol + 3; col++) {
                int value = board[row][col];
                if (value < 1 || value > SIZE || seen[value]) {
                    return false;
                }
                seen[value] = true;
            }
        }
        return true;
    }
}
```

With both “PuzzleGenerator” and “SudokuValidator” discussed, we may now head to the “gui” package.

Unlike the “logic” package, which deals with the backend side of things, the “gui” package is the one responsible for the User Interface, the one that the player would interact with. There are three classes, namely the “SudokuGame”, “StartPopup”, “SudokuGUI” classes.

SudokuGame Class

This class is the controller class, responsible for linking two of the other GUI classes. It launches both the StartPopup and the SudokuGUI classes, which are both the view classes.

```java
package gui;

import javax.swing.*;
```

```
public class SudokuGame {
    public static void main(String[] args) {
        // Launch the StartPopup and handle difficulty selection
        SwingUtilities.invokeLater(() -> {
            new StartPopup(difficulty -> {
                System.out.println("Difficulty selected: " + difficulty);

                // Start the Sudoku GUI with the selected difficulty
                SwingUtilities.invokeLater(() -> new
SudokuGUI(difficulty).setVisible(true));
            });
        });
    }
}
```

StartPopup Class

This class is one of the view classes of the “gui” package. It has two methods as seen below.

The first method, called StartPopup, sets up a “frame” to which the buttons would be placed. To improve the user interface, customized colors were included to make the program more pleasing to the eye.

Java

```
package gui;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class StartPopup extends JFrame {

    public interface DifficultySelectionListener {
        void onDifficultySelected(String difficulty);
    }
```

```
    private DifficultySelectionListener listener;

    public StartPopup(DifficultySelectionListener listener) {
        this.listener = listener;

        // Set up the frame
        setTitle("Sudoku Game");
        setSize(400, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout());
        setResizable(false);

        // Background panel with gradient
        JPanel backgroundPanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                Graphics2D g2d = (Graphics2D) g;
                GradientPaint gradient = new GradientPaint(0, 0, new Color(110,
33, 168), 0, getHeight(), new Color(184, 52, 110));
                g2d.setPaint(gradient);
                g2d.fillRect(0, 0, getWidth(), getHeight());
            }
        };
        backgroundPanel.setLayout(new BoxLayout(backgroundPanel,
BoxLayout.Y_AXIS));
        add(backgroundPanel, BorderLayout.CENTER);

        // Title label
        JLabel titleLabel = new JLabel("SUDOKU");
        titleLabel.setFont(new Font("Arial", Font.BOLD, 32));
        titleLabel.setForeground(Color.WHITE);
        titleLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
        titleLabel.setBorder(BorderFactory.createEmptyBorder(30, 0, 30, 0));
        backgroundPanel.add(titleLabel);

        // Buttons for difficulties
        String[] difficulties = {"Easy", "Medium", "Hard"};
        Color[] buttonColors = {new Color(48, 172, 236), new Color(244, 202,
87), new Color(233, 82, 100)};
        for (int i = 0; i < difficulties.length; i++) {
            JButton button = new JButton(difficulties[i]);
            button.setFont(new Font("Arial", Font.BOLD, 18));
            button.setForeground(Color.WHITE);
```

```
            button.setBackground(buttonColors[i]);
            button.setOpaque(true);
            button.setFocusPainted(false);

            button.setBorder(BorderFactory.createCompoundBorder(
                    BorderFactory.createLineBorder(buttonColors[i], 2),
                    BorderFactory.createEmptyBorder(10, 10, 10, 10)
            ));
            button.setAlignmentX(Component.CENTER_ALIGNMENT);
            button.setMaximumSize(new Dimension(300, 50));
            button.setCursor(new Cursor(Cursor.HAND_CURSOR));

            String difficulty = difficulties[i];
            button.addActionListener(e -> {
                if (listener != null) {
                    listener.onDifficultySelected(difficulty);
                }
                dispose();
            });

            backgroundPanel.add(Box.createVerticalStrut(20));
            backgroundPanel.add(button);
        }

        // Create an "Exit" button
        JButton exitButton = new JButton("Exit");
        exitButton.setFont(new Font("Arial", Font.BOLD, 18));
        exitButton.setForeground(Color.WHITE);
        exitButton.setBackground(new Color(149, 145, 137));
        exitButton.setOpaque(true);
        exitButton.setFocusPainted(false);
        exitButton.setBorder(BorderFactory.createCompoundBorder(
                BorderFactory.createLineBorder(new Color(149, 145, 137), 2),
                BorderFactory.createEmptyBorder(10, 10, 10, 10)
        ));
        exitButton.setAlignmentX(Component.CENTER_ALIGNMENT);
        exitButton.setMaximumSize(new Dimension(300, 50));
        exitButton.setCursor(new Cursor(Cursor.HAND_CURSOR));

        exitButton.addActionListener(e -> {
            System.exit(0); // Exit the program
        });

        // Add the Exit button to the background panel
```

```
        backgroundPanel.add(Box.createVerticalStrut(20));
        backgroundPanel.add(exitButton);

        setLocationRelativeTo(null); // Center the frame on the screen
        setVisible(true);
    }
```

Another method, which is the main method, would handle the difficulty selection by the player.

```Java
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new StartPopup(difficulty -> {
            // Handle difficulty selection
            System.out.println("Selected Difficulty: " + difficulty);
        }));
    }
}
```

SudokuGUI Class

This class is responsible for the puzzle itself. This class has four methods.

The class initially sets up the background of the user interface, as well as the grid that would appear to the player to be inputted on. There are two buttons in this class, which are the “validate” and “return” buttons, which have their own respective functions.

```Java
package gui;

import javax.swing.*;
```

```
import logic.PuzzleGenerator;
import logic.SudokuValidator;
import java.awt.*;

public class SudokuGUI extends JFrame {
    private int[][] board;
    private JTextField[][] cells;

    public SudokuGUI(String difficulty) {
        setTitle("Sudoku Game");
        setMinimumSize(new Dimension(600, 600));
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Set up the gradient background
        JPanel backgroundPanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                Graphics2D g2d = (Graphics2D) g;
                GradientPaint gradient = new GradientPaint(0, 0, new Color(110,
33, 168), 0, getHeight(), new Color(184, 52, 110));
                g2d.setPaint(gradient);
                g2d.fillRect(0, 0, getWidth(), getHeight());
            }
        };
        backgroundPanel.setLayout(new BorderLayout());
        add(backgroundPanel);

        // Center the window
        setLocationRelativeTo(null);

        board = new int[9][9];
        PuzzleGenerator generator = new PuzzleGenerator();
        board = generator.generatePuzzle(difficulty);

        cells = new JTextField[9][9];
        initializeBoard();

        // Create and add the buttons panel
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new FlowLayout());
        buttonPanel.setOpaque(false); // Transparent panel to match background
        backgroundPanel.add(buttonPanel, BorderLayout.SOUTH);
```

```
        // Validate Button
        JButton validateButton = new JButton("Submit");
        validateButton.setFont(new Font("Arial", Font.BOLD, 24));
        validateButton.setBackground(Color.WHITE);
        validateButton.addActionListener(e -> validateBoard());
        buttonPanel.add(validateButton);

        // Return Button
        JButton returnButton = new JButton("Return");
        returnButton.setFont(new Font("Arial", Font.BOLD, 24));
        returnButton.setBackground(Color.WHITE);
        returnButton.addActionListener(e -> {
            dispose();
            SwingUtilities.invokeLater(() -> new StartPopup(newDifficulty ->
new SudokuGUI(newDifficulty).setVisible(true)));
        });
        buttonPanel.add(returnButton);

        // Add the Sudoku board to the center of the background panel
        JPanel boardPanel = new JPanel();
        boardPanel.setLayout(new GridLayout(9, 9));
        for (int row = 0; row < 9; row++) {
            for (int col = 0; col < 9; col++) {
                cells[row][col] = new JTextField();
                cells[row][col].setForeground(Color.RED);

cells[row][col].setBorder(BorderFactory.createLineBorder(Color.BLACK));
                cells[row][col].setHorizontalAlignment(JTextField.CENTER);
                cells[row][col].setFont(new Font("Verdana", Font.BOLD, 24));

                if (board[row][col] != 0) {
                    cells[row][col].setText(String.valueOf(board[row][col]));
                    cells[row][col].setForeground(Color.BLACK);
                    cells[row][col].setFont(new Font("Verdana", Font.PLAIN,
24));
                    cells[row][col].setEditable(false);
                    cells[row][col].setBackground(Color.LIGHT_GRAY);  // Set
background color for given cells
                } else {
                    cells[row][col].setBackground(Color.WHITE);  // Set
background color for editable cells
                }

                boardPanel.add(cells[row][col]);
```

```
            }
        }

        backgroundPanel.add(boardPanel, BorderLayout.CENTER);
    }
```

The initializeBoard method is a method that was already handled in the main constructor.

```Java
    private void initializeBoard() {
        // Already handled inside the `boardPanel` section in the main
constructor
    }
```

The validateBoard method is responsible for validating the user input on the UI itself. It checks if the player correctly inputs the right numbers on the Sudoku grid. If the player got it wrong, then the game repeats; else, the player would be congratulated.

```Java
    private void validateBoard() {
        for (int row = 0; row < 9; row++) {
            for (int col = 0; col < 9; col++) {
                try {
                    String text = cells[row][col].getText();
                    board[row][col] = text.isEmpty() ? 0 :
Integer.parseInt(text);
                } catch (NumberFormatException e) {
                    board[row][col] = 0;
                }
            }
        }

        SudokuValidator validator = new SudokuValidator();
        if (validator.validate(board)) {
            // Custom dialog for valid board
```

```
            showCustomMessage("CONGRATULATIONS! You have completed this
level.", true);
        } else {
            // Custom dialog for invalid board
            showCustomMessage("Oops, please try again!", false);
        }
    }
```

The showCustomMessage is responsible for the custom messages that may appear when using the program.

Java

```
    private void showCustomMessage(String message, boolean isValid) {
        // Create a custom dialog
        JDialog dialog = new JDialog(this, true);
        dialog.setSize(300, 200);
        dialog.setLocationRelativeTo(this); // Center the dialog

        // Create a panel for the message
        JPanel panel = new JPanel();
        panel.setBackground(new Color(182, 52, 111));
        panel.setLayout(new BorderLayout());

        // Add the message label
        JLabel messageLabel = new JLabel(message, SwingConstants.CENTER);
        messageLabel.setFont(new Font("Arial", Font.BOLD, 22));
        messageLabel.setForeground(Color.WHITE);
        panel.add(messageLabel, BorderLayout.CENTER);

        // Add a close button
        JButton closeButton = new JButton("Close");
        closeButton.setFont(new Font("Arial", Font.PLAIN, 16));
        closeButton.addActionListener(e -> dialog.dispose());
        panel.add(closeButton, BorderLayout.SOUTH);

        // Set the dialog's content
        dialog.setContentPane(panel);
        dialog.setVisible(true);
    }
```

The main method is responsible for setting up the new difficulty of the game once the game is done.

```java
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new StartPopup(newDifficulty -> new
SudokuGUI(newDifficulty).setVisible(true)));
    }
}
```

Running the Program without GUI

Recalling the reason why threads are used in this program, these are to ensure that the GUI would work properly as without it, the GUI would not respond to user input. To try something different, the program would then be made this time without GUI, which means that the player would not be able to interact with a user interface, but rather, would use a terminal to run the program.

To run the game without the GUI, the SudokuConsoleGame.java class is included in the repository for the player to play the game. Below is a snippet of the entire program. The class imports the logic package as well as the scanner class to enable user input on the terminal itself. The main method starts the game by calling the startGame method, and that method asks the player on the game difficulty similar to the case with the GUI. If the player inputs a wrong value on the terminal, the terminal will tell the player that the prompt is invalid and has to re-run the game again.

```java
import logic.PuzzleGenerator;
import logic.SudokuValidator;

import java.util.Scanner;

public class SudokuConsoleGame {
```

```
    private static final int SIZE = 9;
    private int[][] board;

    public static void main(String[] args) {
        SudokuConsoleGame game = new SudokuConsoleGame();
        game.startGame();
    }

    public void startGame() {
        Scanner scanner = new Scanner(System.in);
        PuzzleGenerator generator = new PuzzleGenerator();

        System.out.println("Select difficulty (Easy, Medium, Hard): ");
        String difficulty = scanner.nextLine();
        if (!difficulty.matches("Easy|Medium|Hard")) {
            throw new IllegalArgumentException("Difficulty must be Easy,
Medium, or Hard");
        }

        board = generator.generatePuzzle(difficulty);
        play(scanner);
    }
```

The play method then asks the player to input the required information to then validate the game for later. This is where it significantly differs from having a GUI. In this case, instead of the player typing out the numbers on the user interface, the player has to type out the number of the row, the number of the column, and the number itself that would be placed on the specific empty cell. This method calls on other methods to make sure that the program works well.

The number that the player inputs is technically a string, but is converted into an integer by using parseInt to allow the machine to read the user input, thus allowing for the program to see if any of the user input is valid.

The terminal continuously updates the game when the player keeps on inputting the number by printing an updated board with the inputted

number. If in case the player inputs on the wrong cell or inputs a wrong number, the player would be prompted by the program to input a correct value.

Unlike the one with the GUI, this does not need threads to solve the puzzle. Nested if-else statements would check if the program is correct.

```java
    private void play(Scanner scanner) {
        boolean gameOn = true;

        while (gameOn) {
            printBoard();
            System.out.println("Enter your move (row[1-9] col[1-9] value[1-9])
or 'validate' to check solution: ");
            String input = scanner.nextLine();

            if (input.equalsIgnoreCase("validate")) {
                validateBoard();
                gameOn = false; // End the game
            } else {
                try {
                    String[] parts = input.split(" ");
                    int row = Integer.parseInt(parts[0]) - 1;
                    int col = Integer.parseInt(parts[1]) - 1;
                    int value = Integer.parseInt(parts[2]);

                    if (isValidMove(row, col, value)) {
                        board[row][col] = value;
                    } else {
                        System.out.println("Invalid move. Try again.");
                    }
                } catch (Exception e) {
                    System.out.println("Invalid input. Format: row col value");
                }
            }
        }
    }
```

The printBoard method is responsible for printing the sudoku board on the terminal.

```Java
    private void printBoard() {
        for (int row = 0; row < SIZE; row++) {
            for (int col = 0; col < SIZE; col++) {
                System.out.print((board[row][col] == 0 ? "." : board[row][col])
+ " ");
            }
            System.out.println();
        }
    }
```

The validateBoard method validates the board, if the player is correct with the entire board, the program congratulates the player, if the player is incorrect, the program tells the player that the solution is incorrect and would have to repeat the board.

```Java
    private void validateBoard() {
        SudokuValidator validator = new SudokuValidator();
        if (validator.validate(board)) {
            System.out.println("CONGRATULATIONS! You solved the puzzle!");
        } else {
            System.out.println("The solution is incorrect. Try again.");
        }
    }
```

The isValidMove checks if the player inputs the number on an empty cell and also checks if the user inputs a number. If the player is incorrect, then this method is called by the play method and tells the user to try again in inputting on an actual cell.

```Java
    private boolean isValidMove(int row, int col, int value) {
        return row >= 0 && row < SIZE && col >= 0 && col < SIZE && value > 0 &&
value <= SIZE && board[row][col] == 0;
    }
}
```

Downloading the Program

To download the program, simply go to this link: https://github.com/daphnecyrille/CMSC125-MP

Press the “code” button, make sure that it is in HTTPS. Press “copy URL to keyboard”. DO NOT copy anything else while doing this step. At your computer, right click inside of the file of your choice and select “Open in terminal”.

Once the terminal is opened, enter the following:

git clone (Ctrl + V)

Once done, the program is now downloaded.

To actually run the program, make sure that you have an IDE, preferably VSCode or IntelliJ. Inside either IDE, make sure to open the folder called CMSC125-MP. Inside the folder should contain one folder called “src”, and inside that folder should contain two folders for the packages called “gui” and “logic”. Open the class “SudokuGUI” from the “gui” package folder and press the play button to run the program, if you are planning to play with a GUI.

If in case you want to play without the GUI, simply find the SudokuConsoleJava class that is not included in any of the package folders but still well inside the “src” folder. Open the file inside of your IDE and press the play button. Do note that to play the game, the user has to input the necessary inputs on the in-built terminal in the IDE.

The SDK of this program is on Openjdk-23 (Oracle OpenJDK 23), while the language level is SDK default.