Homework 2

数独生成 & 求解器

应用介绍

本 Java 程序可以用于解决或生成一个 9x9 的数独游戏。用户可以选择输入一个数独游戏并得到解答，或者提供提示数目，程序将随机生成一个数独游戏。

使用说明

首先，使用 `javac` 命令编译程序：

```
javac Sudoku.java
```

然后，使用 `java` 命令运行程序：

```
java Sudoku
```

1. 解决数独游戏

运行程序后，输入 1，然后输入数独游戏，空格用 `.` 表示，如下所示：

```
Enter 1 to enter a sudoku and solve, 2 to generate a sudoku:
1
Enter the sudoku:
8 . . . . . . . .
. . 3 6 . . . . .
. 7 . . 9 . 2 . .
. 5 . . . 7 . . .
. . . . 4 5 7 . .
. . . 1 . . . 3 .
. . 1 . . . . 6 8
. . 8 5 . . . 1 .
. 9 . . . . 4 . .
Solution:
8 1 2 7 5 3 6 4 9
9 4 3 6 8 2 1 7 5
6 7 5 4 9 1 2 8 3
1 5 4 2 3 7 8 9 6
3 6 9 8 4 5 7 2 1
2 8 7 1 6 9 5 3 4
5 2 1 9 7 4 3 6 8
4 3 8 5 2 6 9 1 7
7 9 6 3 1 8 4 5 2
```

2. 生成数独游戏

运行程序后，输入 2，然后输入提示数目，程序将随机生成一个数独游戏，如下所示：

```
Enter 1 to enter a sudoku and solve, 2 to generate a sudoku:
2
Enter number of prompts:
30
. 7 5 . 2 . . 8 1
. . . . . 8 . . 2
. . 8 . . . . 6 .
7 . . 2 6 5 . . .
6 . . . 7 . 2 5 .
. . . . . . 8 7 6
2 . . . 9 . 6 4 .
. 5 4 . . 6 3 . 7
. . . . . 2 . . .
```

代码说明

本程序包含 Sudoku 类，包含以下属性：

- `SIZE`：数独游戏的大小，本程序中为 9
- `board`：数独游戏的棋盘，为一个二维数组

本程序包含以下方法：

- `main(String[] args)`：程序入口，用于接收用户输入并调用其他方法
- `print()`：打印当前对象的棋盘
- `scan()`：从控制台输入一个数独游戏
- `generate(int numPrompts)`：生成一个数独游戏，`numPrompts` 为提示数目
- `solve()`：解决数独游戏
- `check()`：检查数独游戏是否合法
- `checkRow(int row)`：检查第 `row` 行是否合法
- `checkCol(int col)`：检查第 `col` 列是否合法
- `checkSquare(int row, int col)`：检查以第 `row` 行、第 `col` 列为左上角的 3x3 方块是否合法
- `clear()`：清空棋盘

算法说明

1. 解决数独游戏

本程序使用回溯法解决数独游戏。首先，程序会从左上角开始，依次尝试填入 1-9，如果填入的数字不合法，则尝试下一个数字，直到找到合法的数字。如果找不到合法的数字，则回溯到上一个空格，重新填入数字。如果所有空格都填满了数字，则数独游戏解决成功。

2. 生成数独游戏

本程序使用回溯法生成数独游戏。由于我们在解决数独游戏时，每次都是从左上角开始，以随机的顺序尝试填入数字，因此我们可以先生成一个随机的数独游戏，然后随机地删除一些数字，直到达到指定的提示数目。

不足与改进

- 尽管本程序有一定的异常处理，但是出现异常时只会提示 `Invalid input`，而不会提示具体的错误信息，这对于用户来说不够友好。
- 本程序使用命令行界面进行交互，用户体验不够好，可以考虑使用图形界面。
- 将 `main` 方法与 `Sudoku` 类分离，可以使得程序更加清晰。
- 生成的数独游戏可能有多个解。在求解数独游戏时，如果找到一个解，程序就会停止，而不会继续寻找其他解。

附录

代码如下：

```
import java.util.Scanner;
import java.util.Random;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Sudoku {

    public final static int SIZE = 9;       // Size of the Sudoku board
    int[][] board = new int[SIZE][SIZE];    // The Sudoku board

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Sudoku sudoku = new Sudoku();
        System.out.println("Enter 1 to enter a sudoku and solve, 2 to generate a
sudoku: ");
        try {
            int choice = scanner.nextInt();
            if (choice == 1) {  // Solve a Sudoku
                sudoku.scan();
                if (sudoku.solve()) {
                    System.out.println("Solution:");
                    sudoku.print();
                } else {
                    System.out.println("No solution");
                }
            } else if (choice == 2) {   // Generate a Sudoku
                System.out.println("Enter number of prompts: ");
                int numPrompts = scanner.nextInt();
                if (numPrompts < 0 || numPrompts > 81) {
                    System.out.println("The number of prompts must be between 0
and 81");
                    scanner.close();
                    return;
                }
                sudoku.generate(numPrompts);
                sudoku.print();
            } else {
                System.out.println("Invalid input");
            }
        } catch (Exception e) {
```

```
            System.out.println("Invalid input");
        }
        scanner.close();
    }

    /**
     * Prints the Sudoku board.
     */
    void print() {
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                System.out.print(board[i][j] == 0 ? "." :
Integer.toString(board[i][j]));
                System.out.print(' ');
            }
            System.out.println();
        }
    }

    /**
     * Scans the Sudoku board from the console.
     * @throws Exception If the input is invalid.
     */
    void scan() throws Exception {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter the sudoku: ");
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                String input = scanner.next();
                if (input.equals(".")) {    // Empty cell
                    board[i][j] = 0;
                } else {
                    board[i][j] = Integer.parseInt(input);
                    // Check if the input is valid
                    if (board[i][j] < 1 || board[i][j] > SIZE) {
                        scanner.close();
                        throw new Exception();
                    }
                }
            }
        }
        scanner.close();
    }

    /**
     * Generates a Sudoku board with the given number of prompts.
     * @param numPrompts The number of prompts to generate.
     */
    void generate(int numPrompts) {
        Random rand = new Random();

        // Generate a random solved Sudoku board
        solve();

        // Remove random entries from the board
```

```
        for (int i = 0; i < 81 - numPrompts; i++) {
            int row = rand.nextInt(SIZE);
            int col = rand.nextInt(SIZE);
            // Make sure the cell is not empty
            while (board[row][col] == 0) {
                row = rand.nextInt(SIZE);
                col = rand.nextInt(SIZE);
            }
            board[row][col] = 0;
        }
    }

    /**
     * Solves the Sudoku board.
     * @return True if the board is solved, false otherwise.
     */
    boolean solve() {
        // Generate a random order of numbers to try
        List<Integer> nums = new ArrayList<Integer>();
        for (int i = 1; i <= SIZE; i++) {
            nums.add(i);
        }
        Collections.shuffle(nums);

        // Find the next empty cell
        int nexti = 0, nextj = 0;
        boolean found = false;
        for (int i = 0; i < SIZE && !found; i++) {
            for (int j = 0; j < SIZE && !found; j++) {
                if (board[i][j] == 0) {
                    nexti = i;
                    nextj = j;
                    found = true;
                }
            }
        }
        if (!found) {   // No empty cells, check if the complete board is valid
            return check();
        }

        // Try each number in the cell
        for (int i = 0; i < SIZE; i++) {
            board[nexti][nextj] = nums.get(i);
            if (!checkRow(nexti) || !checkCol(nextj) || !checkSquare(nexti -
nexti % 3, nextj - nextj % 3)) {
                // Number is invalid, try the next number
                continue;
            }
            if (solve()) {
                return true;
            }
        }

        // No solution found, backtrack
        board[nexti][nextj] = 0;
```

```
        return false;
    }

    /**
     * Checks if the Sudoku board is valid.
     * @return True if the board is valid, false otherwise.
     */
    boolean check() {
        // Check each row, column, and square
        for (int i = 0; i < SIZE; i++) {
            if (!checkRow(i) || !checkCol(i)) {
                return false;
            }
        }
        for (int i = 0; i < SIZE; i += 3) {
            for (int j = 0; j < SIZE; j += 3) {
                if (!checkSquare(i, j)) {
                    return false;
                }
            }
        }
        return true;
    }

    /**
     * Checks if the given row is valid.
     * @param row The row to check.
     * @return True if the row is valid, false otherwise.
     */
    boolean checkRow(int row) {
        boolean[] nums = new boolean[SIZE];
        for (int i = 0; i < SIZE; i++) {
            if (board[row][i] != 0) {
                if (nums[board[row][i] - 1]) {  // Number already exists in row
                    return false;
                }
                nums[board[row][i] - 1] = true;
            }
        }
        return true;
    }

    /**
     * Checks if the given column is valid.
     * @param col The column to check.
     * @return True if the column is valid, false otherwise.
     */
    boolean checkCol(int col) {
        boolean[] nums = new boolean[SIZE];
        for (int i = 0; i < SIZE; i++) {
            if (board[i][col] != 0) {
                if (nums[board[i][col] - 1]) {  // Number already exists in
column
                    return false;
                }
```

```
                nums[board[i][col] - 1] = true;
            }
        }
        return true;
    }

    /**
     * Checks if the given square is valid.
     * @param row The row of the square to check.
     * @param col The column of the square to check.
     * @return True if the square is valid, false otherwise.
     */
    boolean checkSquare(int row, int col) {
        boolean[] nums = new boolean[SIZE];
        for (int i = row; i < row + 3; i++) {
            for (int j = col; j < col + 3; j++) {   // Check each cell in the
square
                if (board[i][j] != 0) {
                    if (nums[board[i][j] - 1]) {
                        return false;
                    }
                    nums[board[i][j] - 1] = true;
                }
            }
        }
        return true;
    }

    /**
     * Clears the Sudoku board.
     */
    void clear() {
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                board[i][j] = 0;
            }
        }
    }

}
```