Homework 1: Sudoku Programming

1. 程序介绍

本程序为数独程序，实现了要求的两大功能，Sudoku生成器和Sudoku求解器。代码位于文件 `/src` 目录下的Sudoku文件中，需要通过编译命令：

```
javac SudokuGame.java
```

编译成可执行文件，

然后输入命令：

```
java Sudoku
```

方可执行。

成功执行之后，终端会输出提示 `Please enter the operation of Sukudo (Solve/Generate)`

之后用户要输入 `Solve` 或者 `Generate` 来启动Sudoku求解器或者Sudoku生成器。

- 输入`Solve`

  终端会输出提示 `Please enter the Sukudo:`，用户需要输入一个 9×9 的矩阵来代表数独，其中，待填的格子用0表示。

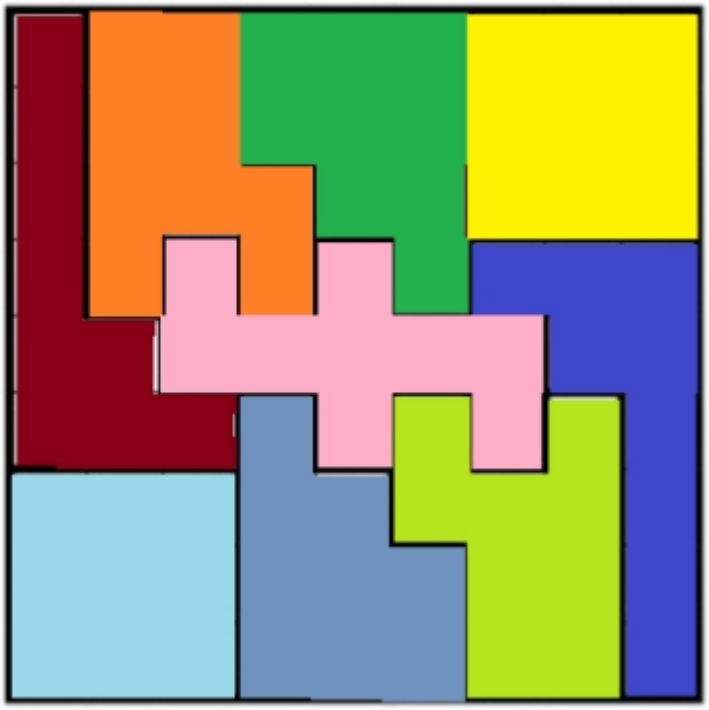
  一个合法的输入样例如下：

  ```
  0 2 0 0 0 0 0 0 0
  2 0 0 0 0 0 0 0 9
  9 5 0 0 4 0 0 0 0
  0 9 3 0 0 0 0 0 0
  0 7 0 0 0 0 1 0 0
  0 6 0 0 0 0 0 0 0
  5 4 2 9 1 7 3 0 0
  0 0 0 6 0 0 0 5 4
  6 0 0 0 0 0 0 0 0
  ```

  输入数独矩阵并且回车后，终端会输入提示 `Please enter the mask matrix:`，用户需要输入一个 9×9 的矩阵来代表掩码矩阵，其中，不同的区域由0~8的数字来区别，以下为合法的输入样例：

```
0 1 1 2 2 2 3 3 3
0 1 1 2 2 2 3 3 3
0 1 1 1 2 2 3 3 3
0 1 4 1 4 2 5 5 5
0 0 4 4 4 4 4 5 5
0 0 0 7 4 8 4 8 5
6 6 6 7 7 8 8 8 5
6 6 6 7 7 7 8 8 5
6 6 6 7 7 7 8 8 5
```

它代表的数独的区域划分如下：



输入掩码矩阵并且回车后，程序会自动给出答案。对于上述的输入样例，程序的解答为：

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

- 输入Generate

  终端会输出提示 `Please enter the number of hints:`，用户需要输入提示数（1~81），即数独中已经填好的格子的个数，例如 22 。

  输入提示数并回车后，终端会输出提示 `Please enter the mask matrix:`，用户需要输入一个 9×9 的掩码矩阵，以下为合法的输入样例：

```
0 1 1 2 2 2 3 3 3
0 1 1 2 2 2 3 3 3
0 1 1 1 2 2 3 3 3
0 1 4 1 4 2 5 5 5
0 0 4 4 4 4 4 5 5
0 0 0 7 4 8 4 8 5
6 6 6 7 7 8 8 8 5
6 6 6 7 7 7 8 8 5
6 6 6 7 7 7 8 8 5
```

输入掩码矩阵并回车后，终端会输出提示 `Do you need the sukudo must have solution? (Yes/No)`， 表明用户是否要确保生成的数独一定有解，**需要注意的是，若选择确保一定有解，则生成数独将要花费大约2min的时间，请耐心等待；若选择不确保有解，则提示数越少，生成的数独有解的概率越大，相应的求解时间也越长。。**

当输入 `Yes` 或者 `No` 后，程序会生成数独（含有随机性，每次结果不一定一样），

以下是选择Yes后的一个可能的输出样例：

```
0 0 0 3 0 0 0 5 0
0 3 0 1 5 9 7 0 0
0 0 0 0 0 4 0 0 8
0 0 0 7 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 4 0 0 8 0 0
1 0 0 0 3 0 5 0 0
0 0 0 0 6 5 0 1 7
0 0 0 8 0 0 4 0 3
```

通过本程序的数独求解器可以求出它的一个解为：

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

以下是选择No后的一个可能都输出样例：

```
7 0 0 3 0 0 9 0 0
0 0 5 0 0 0 4 0 0
0 0 0 0 0 0 8 7 0
0 1 0 0 4 0 0 0 0
0 5 0 0 0 0 0 9 0
0 0 8 0 0 7 1 0 0
5 8 0 9 0 0 0 0 2
6 0 4 0 3 8 0 0 0
0 0 0 0 5 0 0 0 0
```

通过本程序的数独求解器可以判断其无解：

```
No solutions!
```

2. 代码说明

主类SudokuGame：

```
public class SudokuGame {
    public static void main(String argv[]) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Please enter the operation of Sudoku (Solve/Generate)");
        String ans = scanner.nextLine();
        if(ans.equals("Generate")) {
            Sudoku sud = new Sudoku();
            sud.generateSudoku();
        }
        else if(ans.equals("Solve")) {
            Sudoku suk = new Sudoku();
            suk.solveSudoku();
        }
        scanner.close();
    }
}
```

负责程序的启动和生成器、求解器的选择。

position类

```
class position {
    int x;
    int y;
    position (int x, int y) {
        this.x = x;
        this.y = y;
    }
    public boolean equals(Object obj) {
        if(obj == this) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if(obj instanceof position) {
            position pos = (position) obj;
            if(this.x == pos.x && this.y == pos.y)
                return true;
        }
```

```
            return false;
        }
    }
```

保存格子坐标的二元组，辅助对掩码矩阵不同区域的分类。

Sudoku类：

```
class Sudoku {
    private int prompt_num;
    private int mask_matrix[][];
    private int value_matrix[][];
    private int puzzle_matrix[][];
    private int fixed_matrix[][];
    private ArrayList<position> pos_list[];
    Sudoku() {...}
    void generateSudoku() {...}
    void solveSudoku() {...}
    boolean checkNum(int i, int j, int num) {...}
    boolean solve() {...}
    void randomFill() {...}
    void swapNum() {...}
    void printValueMatrix() {...}
    void printPuzzleMatrix() {...}
    void randomFillNumber() {...}
```

负责数独实例的创建，求解和生成。

- 类成员
  - prompt_num：储存提示数。
  - mask_matrix：储存掩码矩阵。
  - value_matrix：储存数独的解。
  - puzzle_matrix：储存生成的数独。
  - fixed_matrix：储存数独预先填好的位置。
  - pos_list：将掩码相同的位置储存在List中，将不同的List按照掩码值储存在pos_list中。
- 类方法
  - Sudoku()

    构造方法，初始化类成员。
  - printValueMatrix()

    打印ValueMatrix
  - printPuzzleMatrix()

    打印PuzzleMatrix
  - generateSudoku()

    数独生成器，读入掩码矩阵和提示数。

若用户选择确保有解，则根据掩码矩阵，通过回溯算法找出一组数独解（详见solve()方法），然后将进行数字随机交换（详见swapNum()方法），最后在数独解中随机挖空，直到只有提示数个填充好的格子（详见randomFill()方法）。

若用户选择不确保有解，则在 9×9 的矩阵里按照数独规则，随机填充提示数个数字（详见randomFillNumber()方法）。

```
    void generateSudoku() {
        try {
            Scanner scanner = new Scanner(System.in);
            System.out.println("Please enter the number of hints: ");
            prompt_num = scanner.nextInt();
            System.out.println("Please enter the mask matrix: ");
            for (int i=0; i<9; i++) {
                for (int j=0; j<9; j++) {
                    int mask = scanner.nextInt();
                    position pos = new position(i, j);
                    pos_list[mask].add(pos);
                    mask_matrix[i][j] = mask;
                }
            }
            scanner.nextLine();
            System.out.println("Do you need the sukudo must have solution? (Yes/No)");
            String ans = scanner.nextLine();
            scanner.close();
            //generate
            System.out.println("Generating...");
            if(ans.equals("Yes")) {
                if(solve()) {
                    //number swap
                    swapNum();
                    //random clean
                    randomFill();
                    System.out.println("Successfully generated!");
                }
                else
                    throw new Exception("This mask matrix has no solutions!");
            }
            else if(ans.equals("No")) {
                randomFillNumber();
            }

        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Puzzle: ");
        printPuzzleMatrix();
    }
```

- solve()

  求解数独，通过回溯算法，遍历 9×9 个格子，每遍历到一个格子，判断可以填入该格子的数字（详见checkNum()方法），如果没有可以填入的数字，就回溯到上一个格子，再选择一个其他可填入的数字填入。当遍历所有格子，就求出了数独的一个解。

  需要注意，有些格子一开始就填充好了固定的数字，对于这些格子，不论是向前遍历还是向后回溯都要忽略。

  此外，如果发现回溯到了第一个格子，发现没有可以填充的数字时，程序即可判断数独无解。

```
    boolean solve() {
        boolean back = false;
        for(int i=0; i<9; i++) {
            for(int j=0; j<9; j++) {
                if(back && fixed_matrix[i][j] != 0) {
                    if(j>0) {
                        j = j - 2;
                        back = true;
                    }
                    else if(i>0) {
                        back = true;
                        i--;
                        j = 7;
                    }
                    else {
                        return false;
                    }
                    continue;
                }
                if(fixed_matrix[i][j] != 0) {
                    value_matrix[i][j] = fixed_matrix[i][j];
                    continue;
                }
                boolean changed = false;
                int k = back? value_matrix[i][j] + 1: 1;
                for(; k<10; k++) {
                    if(checkNum(i, j, k)) {
                        value_matrix[i][j] = k;
                        changed = true;
                        back = false;
                        break;
                    }
                }
                if(!changed) {
                    value_matrix[i][j] = 0;
                    if(j>0) {
                        j = j - 2;
                        back = true;
                    }
                    else if(i>0) {
                        back = true;
                        i--;
                        j = 7;
                    }
                    else {
                        return false;
                    }
                }
            }
        }
        return true;
    }
```

- checkNum()

  检查填入的数字是否符合规则，即同一行、同一列和同一掩码的区域内没有和它相同的数字。

```
boolean checkNum(int i, int j, int num) {
        for(int m=0; m<9; m++) {
```

```
            if(value_matrix[m][j] == num)
                return false;
        }
        for(int n=0; n<9; n++) {
            if(value_matrix[i][n] == num)
                return false;
        }
        int mask = mask_matrix[i][j];
        for(position p : pos_list[mask]) {
            if(value_matrix[p.x][p.y] == num)
                return false;
        }
        return true;
    }
```

- swapNum()

  在生成数独时，对求解出的数独进行数字交换，即先生成一个1~9的随机序列，然后进行分组：该序列的第一个数对应最后一个数，第二个数对应倒数第二个数...以此类推，这样就生成了一个数字的映射表A，随后遍历求解出的数独value_matrix，对于某个 value_matrix[i][j]，将其替换为 A[value_matrix[i][j]]。这样就随机地打乱了数独，且仍然符合数独的规则。

```
void swapNum() {
        long seed = System.currentTimeMillis();
        Random random = new Random(seed);
        ArrayList<Integer> numbers = new ArrayList<>();
        for (int i = 1; i <= 9; i++) {
            numbers.add(i);
        }
        Collections.shuffle(numbers, random);
        int arr[] = new int[10];
        for(int i=0; i<9; i++) {
            arr[numbers.get(i)] = numbers.get(8-i);
        }
        for(int i=0; i<9; i++) {
            for(int j=0; j<9; j++) {
                value_matrix[i][j] = arr[value_matrix[i][j]];
            }
        }
    }
```

- randomFill()

  对生成的完整的数独进行随机挖空，直到只有提示数个格子有数字。

```
void randomFill() {
        for(int i=0; i<prompt_num; i++) {
             int row, col;
             long seed = System.currentTimeMillis();
             Random random = new Random(seed);
             do {
                row = random.nextInt(9);
                col = random.nextInt(9);
            } while (puzzle_matrix[row][col] != 0);
            puzzle_matrix[row][col] = value_matrix[row][col];
        }
    }
```

- randomFillNumber()

  对空白的数独随机填充提示数个数字，填入时要满足数独的规则，若随机生存的位置上的数字不满足，则重新生成。

```
void randomFillNumber() {
        for(int i=0; i<prompt_num; i++) {
             int row, col, value;
             long seed = System.currentTimeMillis();
             Random random = new Random(seed);
             boolean flag = false;
             do {
                 flag = false;
                 row = random.nextInt(9);
                 col = random.nextInt(9);
                 value = random.nextInt(9) + 1;
                 if(puzzle_matrix[row][col] != 0)
                     flag = true;
                 for(int m=0; m<9; m++) {
                     if(puzzle_matrix[m][col] == value)
                     {
                         flag = true;
                         break;
                     }
                 }
                 for(int n=0; n<9; n++) {
                     if(puzzle_matrix[row][n] == value)
                     {
                         flag = true;
                         break;
                     }
                 }
                 int mask = mask_matrix[row][col];
                 for(position p : pos_list[mask]) {
                     if(puzzle_matrix[p.x][p.y] == value) {
                         flag = true;
                         break;
                     }
                 }
                 if(!flag)
                     puzzle_matrix[row][col] = value;
             } while(flag);
        }
    }
```

- solveSudoku()

  根据输入的数独和掩码矩阵，求解数独，若有解，则输出答案，否则输出无解。

```
void solveSudoku() {
        try {
            Scanner scanner = new Scanner(System.in);
            System.out.println("Please enter the Sukudo: ");
            for(int i=0; i<9; i++) {
                for(int j=0; j<9; j++) {
                    value_matrix[i][j] = scanner.nextInt();
                    fixed_matrix[i][j] = value_matrix[i][j];
                }
```

```
            }
            System.out.println("Please enter the mask matrix: ");
            for (int i=0; i<9; i++) {
                for (int j=0; j<9; j++) {
                    int mask = scanner.nextInt();
                    position pos = new position(i, j);
                    pos_list[mask].add(pos);
                    mask_matrix[i][j] = mask;
                }
            }
            scanner.close();
            System.out.println("Solving...");
            if(solve()) {
                System.out.println("Answer: ");
                printValueMatrix();
            }
            else
                throw new Exception("No solutions!");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

    }
```