```
/**
 *
 * Author: Lukas Ottenhof
 * ID: 1734382
 * Class: AUSCI 235
 * Date: 2023-03-24
 *
 * This is a file that will solve a maze constructed with chars in a text file.
 * this program will find a direct path through the maze and print the maze
 * with the direct path marked by '.'
 *
 * it uses the following methods,
 *
 * makeMazeArray --> uses findingMazeDimensions to make an array the size of the
 * maze and put the maze into a char[][] array
 *
 * findingMazeDimensions --> this is used to find the dimensions of the maze
 *
 * findEntry --> finds the entry point of the maze
 *
 * checkIfSolvable --> checks to see if the maze is solvable
 *
 * treavelingThrough --> creates stacks containing the direct path. It uses
 * getNextBlank to get points in the path and printStack to put '.' in the array
 * at the points that are in the direct path indicated by the coordinates stored
 * in the stacks
 *
 * getNextBlank --> finds the next blank after the one that was just put in the
 * stack and makes sure its also not one of the previous ones so we don't go
 * backwards
 *
 * printStack --> puts the direct path in the array. the direct path is stored
 * as coordinates in the stacks
 *
 * printMaze --> this is used to print the solved maze array to the user
 */
import java.util.Scanner;
import java.util.Stack;
import java.io.File;
import java.io.FileNotFoundException;

public class MazeSolver {

    //files for testing
    private static final String Maze1 = "C:\\Users\\otten\\OneDrive\\Documents"
            + "\\NetBeansProjects\\MazeSolver\\src\\main\\java\\Maze1.txt";
    private static final String Maze2 = "C:\\Users\\otten\\OneDrive\\Documents"
            + "\\NetBeansProjects\\MazeSolver\\src\\main\\java\\Maze2.txt";
    private static final String Maze3 = "C:\\Users\\otten\\OneDrive\\Documents"
            + "\\NetBeansProjects\\MazeSolver\\src\\main\\java\\Maze3.txt";
    private static final String Maze4 = "C:\\Users\\otten\\OneDrive\\Documents"
            + "\\NetBeansProjects\\MazeSolver\\src\\main\\java\\Maze4.txt";
    private static final String CantSolveTest = "C:\\Users\\otten\\OneDrive\\"
            + "Documents\\NetBeansProjects\\MazeSolver\\src\\main\\java\\"
            + "CantSolve.txt";

    /**
     * in main, the file will be created, then a method that turns the file into
     * an array will be used, then a method to check if it is solvable is
     * called, if it isn't solvable the program will end and the user will be
     * told the maze isn't solvable but if it is solvable the method that finds
     * and returns the array with the direct path will be called and if the
     * direct path was found the maze with the direct path will be printed
     *
     * @param args
     */
    public static void main(String[] args) {
        File myFile = new File(Maze1); //the maze file

        char[][] mazeArray = makeMazeArray(myFile);//build array that will
        //contain the maze

        if (!checkIfSolvable(mazeArray)) {//checking if maze is solvable
            System.out.println("This maze is not solvable");
        } else {
            mazeArray = travelingThrough(mazeArray);//if the maze has an exit we
            //will try to solve

            if (mazeArray[0][0] == 'N') {
                System.out.println("This maze is not solvable");//if there is
                //a 'N' in the first spot this was a a singal telling us that
                //it wasnt solvable because there was a blockage somewhere in
                //the maze

            } else {
                printMaze(mazeArray);//printing the maze with direct path if it
                //was solved
            }
        }

    }//end of main

    /**
     * This method takes a file and uses another method to find its width and
     * height to create an array containing the maze in the given file. The
     * dimensions of the new array will be taken from an array calculated by a
     * different method where the first value is the number of rows and the
     * second is the number of cols. once an array has been created with the
     * dimensions that the findingMazeDimensions method found, this method will
     * go row by row through the file containing maze and it will split each row
     * into a char array and then put each char in this array into the
     * corresponding row in the mazeArray.
     *
     *
     * @param myFile the file containing the maze we want to solve
     * @return the char array containing the maze will be returned
     */
    public static char[][] makeMazeArray(File myFile) {
        //array that will contiain calculated dimensions
        int[] dimensions = findingMazeDimensions(myFile);

        //making an array with the calculated dimensions storred in the array
        //above
        char[][] mazeArray = new char[dimensions[0]][dimensions[1]];

        //making a scanner to put the file into a 2d char array
        Scanner scannerToPutFileInArray;
        try {
            scannerToPutFileInArray = new Scanner(myFile);
        } catch (FileNotFoundException e) {
            System.out.println("ERROR");
            return null;
        }

        //going row by row of the file spliting each row into a char array and
        //putting this new char array into the rows of our array
        while (scannerToPutFileInArray.hasNextLine()) {
            for (int row = 0; row < mazeArray.length; row++) {
                //new temp array containing the row in a char array
                char[] temp = scannerToPutFileInArray.nextLine().toCharArray();
                for (int col = 0; col < mazeArray[row].length; col++) {
                    mazeArray[row][col] = temp[col];
                }
            }
        }//end of while
        return mazeArray;//return completed array
    }//end of makeMazeArray

    /**
     * this method calculates the dimensions of our array. it does this by
     * taking the file as a parameter and creating a scanner to move through it.
     * To find the cols the scanner takes the first row and calculates its
     * length. then to find its rows it gets the next row until there is no next
     * row and adds 1 to a counter containing the number of rows the loop that
     * does this starts at 1 because we already got the first row to find the
     * number of cols. these calculated dimensions are returned in an array that
     * contains the number of rows in col 0, and the number of cols in col 1.
     * these dimensions are returned in an array because we cant return 2 ints
     *
     * @param myFile file we are finding the dimensions of
     * @return the dimensions stored in an array will be returned
     */
    public static int[] findingMazeDimensions(File myFile) {
        int[] dimensions = new int[2];//array that will stoer dimensions

        //making a scanner that will be used to get the dimensions
        Scanner scannerToFindSize;
        try {
            scannerToFindSize = new Scanner(myFile);
        } catch (FileNotFoundException e) {
            System.out.println("ERROR");
            return null;
        }

        //length of the first row will be counted to get number of cols
        int totalCols = (scannerToFindSize.nextLine().length());

        //to get rows in file
        int totalRows;
        for (totalRows = 1; scannerToFindSize.hasNextLine(); totalRows++) {
            scannerToFindSize.nextLine();
        }

        scannerToFindSize.close();//closing the scanner

        dimensions[0] = totalRows;
        dimensions[1] = totalCols;
        return dimensions;//returning the array that has the dimensions stored
        //in it
    }//end of finding dimensions

    /**
     * this method finds the entry. it does this by going row by row in the
     * first col and looking for the opening, ' ', spot. The row that the entry
     * is in will be returned. We don't need the col since the entry will always
     * be in col 0. If there is no entry -1 will be returned by default
     *
     * @param mazeArray array containing the maze we are looking at
     * @return the row containing the entry will be returned
     */
    public static int findEntry(char[][] mazeArray) {
        int entry = -1; //if this is returned there is no entry
        for (int row = 0; row < mazeArray.length; row++) {
            if (mazeArray[row][0] == ' ') {
                return row; //row containing entry
            }
        }
        return entry;//this will only happen if there is no entry
    }//end of find entry

    /**
     * this method checks to see if the maze is solvable. it does this by
     * looking to see if there is an exit. the method finds if there is an exit
     * by looking in the lase col of our array and trying to find if there is a
     * ' ' spot. if there is a ' ' in the last row true will be returned to show
     * that it is solvable, otherwise false will be returned to show that the
     * maze is not solvable
     *
     * @param mazeArray array containing the maze we are looking at
     * @return a boolean will be returned that says true of false to show if the
     * maze is solvable
     */
    public static boolean checkIfSolvable(char[][] mazeArray) {
        //going row by row looking for exit
        for (int row = 0; row < mazeArray.length - 1; row++) {
            //every row in last col
            if (mazeArray[row][mazeArray[0].length - 1] == ' ') {
                return true;//there is a entry
            }
        }
        return false;//there is no entry
    }//end of check if solvable

    /**
     * this is the "main" method that is used to solve the maze. it creates
     * stacks that will contain the direct path and variables that have the row
     * and col last put on the stack, and the ones put on the stack 1 before the
     * current ones. this method starts by pushing the starting position onto
     * the stack. From here I push coordinates onto the stack by finding the
     * coordinates of the next ' ' spot after the previous spot (previous spot
     * is stored in the currentRow and currentCol variables) that was pushed
     * into the stack, (i used 2 stacks, one for row one for col) and if they
     * lead to a dead end I pop back to the last spot where there was a choice
     * in what direction to go, blocking off the spots being popped that lead to
     * a dead end with 'w', so it won't try them again, and try a direction that
     * hasn't been blocked off. This continues until the last position put into
     * the stacks is in the same col as the exit. If there is no solution
     * because all paths lead to a dead end, 'N' will be put into the first
     * position in the array to signal back to main that it is not solvable
     *
     * @param mazeArray
     */
    public static char[][] travelingThrough(char[][] mazeArray) {
        //these are the points that had been pushed on the stack last
        int currentCol = 0;
        int currentRow = 0;

        int[] rowColArray = new int[3]; //an array that will contain the
        //cordinates of the newx ' ' spot found by the getNextBlank method.
        //if there is a -1 in col 2, it has hit a dead end

        //these are the stacks containing the rows and cols of the direct path.
        //together they hold the coordinates of the direct path
        Stack<Integer> colStack = new Stack<>();
        Stack<Integer> rowStack = new Stack<>();
        currentRow = findEntry(mazeArray); //finds the starting row
        //pushing the starting points onto the stack
        colStack.push(currentCol);
        rowStack.push(currentRow);

        //cordinate before last put onto stack, these are kept so we dont
        //start going backwards through the maze at points weve already been
        int secondLastRow = 0;
        int secondLastCol = 0;

        //while hasnt reaches the side with an exit continue
        while (currentCol < mazeArray[0].length - 1) {

            //find the next blank, coordinates of this spot are stored in
            //rowColArray
            rowColArray = getNextBlank(mazeArray, currentRow, currentCol,
                    rowColArray, secondLastRow,
                    secondLastCol);

            //if there is a -1 in col 3 of the row col array this means we hit a
            //dead end and need to move back until there is a new direction we
            //can go
            if (rowColArray[2] == -1) {
                //go back until no longer at a dead end
                while (rowColArray[2] == -1) {
                    mazeArray[currentRow][currentCol] = 'w'; //blocking off the
                    //dead end so it doesnt try that spot again

                    //moving backwards
                    //setting the last spot entered to be the second last
                    currentRow = (int) rowStack.pop();
                    currentCol = (int) colStack.pop();

                    //if we poped all the way back to the start and could not
                    //find a path that works there is a blockage in the maze
                    //that makes it unsolvable, and the stacks will be empty.
                    //setting this N tells main it is unsolvable
                    if (colStack.empty()) {
                        mazeArray[0][0] = 'N';
                        return mazeArray;
                    }

                    secondLastRow = (int) rowStack.peek();//getting the second
                    secondLastCol = (int) colStack.peek();//last coordinate

                    //try to get a next spot, if its no longer dead end loop
                    //will end, if it is a dead end the loop will continue
                    //moving us backwards
                    rowColArray = getNextBlank(mazeArray, currentRow,
                            currentCol, rowColArray,
                            secondLastRow, secondLastCol);
                }

                //after the loop has been completed update the stack to contain
                //the condinate that was popped back to last
                rowStack.push(currentRow);//pushing the last into the stack
                colStack.push(currentCol);
            }//end of if

            //if the next spot was not a dead end the if will be skipped

            //set the second last spot to the last spot on the stack
            secondLastRow = (int) rowStack.peek();//getting the second last
            secondLastCol = (int) colStack.peek();

            //update the current varables to contain the cordiante of the new
            //blank
            currentRow = rowColArray[0];
            currentCol = rowColArray[1];

            //push the new blank onto the stack
            rowStack.push(currentRow);
            colStack.push(currentCol);

        }//end of loop that finds the direct path
        mazeArray = printStack(rowStack, colStack, mazeArray);//putting the
        //direct path into the array and removing all 'w'
        return mazeArray;//returning the complete maze
    }//end of traveling through

    /**
     * this method is used to get the next blank spot.it returns an array with
     * the coordinates of the next blank spot.it does this by "looking" right,
     * down, up and left of the last blank spot for the next blank spot. it does
     * this by looking in a direction from the previous spot and checking if it
     * is also a blank spot and is not a spot we have been to. if there is no
     * spot that meets these requirement then the spot were were trying to move
     * from will be returned along with -1 on the second column
     *
     * @param mazeArray the array containing the maze its trying to solve
     * @param currentRow the row we are looking from
     * @param currentCol the col we are looking from
     * @param oldRowColArray array containing old points
     * @param lastRow the last row, needed so we don't go backwards
     * @param lastCol the last col, needed so we don't go backwards
     * @return an array containing the coordinates of the next spot will be
     * returned. it can also signal that a dead end has been hit and we need to
     * move backwards
     */
    public static int[] getNextBlank(char[][] mazeArray, int currentRow,
            int currentCol, int[] oldRowColArray, int lastRow, int lastCol) {

        //getting the old coordinates from the array
        int oldRow = oldRowColArray[0];
        int oldCol = oldRowColArray[1];

        //creating a new array that will store the next blanks coordinates
        int[] rowColArray = new int[3];

        //going right by adding 1 to the col, making sure its a ' ' and that
        //we havent been there
        if (mazeArray[currentRow][currentCol + 1] == ' ' && currentCol
                + 1 != oldCol && currentCol + 1 != lastCol) {
            //if the requierments are met put the coordinates in the array and
            //return the array
            rowColArray[0] = currentRow;
            rowColArray[1] = currentCol + 1;
            return rowColArray;
        }

        //going down by adding 1 to the row, making sure its a ' ' and that
        //we havent been there
        if (mazeArray[currentRow + 1][currentCol] == ' ' && currentRow
                + 1 != oldRow && currentRow + 1 != lastRow) {
            //if the requierments are met put the coordinates in the array and
            //return the array
            rowColArray[0] = currentRow + 1;
            rowColArray[1] = currentCol;
            return rowColArray;
        }
        //going up by subtracting 1 from row, making sure its a ' ' and that
        //we havent been there
        if (mazeArray[currentRow - 1][currentCol] == ' ' && currentRow - 1
                != oldRow && currentRow - 1 != lastRow) {
            //if the requierments are met put the coordinates in the array and
            //return the array
            rowColArray[0] = currentRow - 1;
            rowColArray[1] = currentCol;
            return rowColArray;
        }
        //going left by subtracting 1 from col, making sure its a ' ' and that
        //we havent been there
        if (mazeArray[currentRow][currentCol - 1] == ' ' && currentCol - 1
                != oldCol && currentCol - 1 != lastCol) {
            //if the requierments are met put the coordinates in the array and
            //return the array
            rowColArray[0] = currentRow;
            rowColArray[1] = currentCol - 1;
            return rowColArray;
        } else {
           //if there was no new blank that can be reached send back the current
            //position and set the last col to -1 to singal we need to move back
            rowColArray[0] = oldRow;
            rowColArray[1] = oldCol;
            rowColArray[2] = -1; //signaling that weve hit a dead end
            return rowColArray;
        }
    }//end of get next blank

    /**
     * this method is used after we have the complete stacks that contain the
     * direct path. this method pops the stacks getting the coordinates of the
     * correct path putting '.' at these coordinates in the array until the
     * stacks are empty. then all 'w' are cleared from the array
     *
     * @param rowStack stack containing all rows of the direct path
     * @param colStack stack containing all cols of the direct path
     * @param mazeArray the array containing the maze we want to solve
     * @return the updated mazeArray than now contains the direct path will be
     * returned
     */
    public static char[][] printStack(Stack rowStack, Stack colStack,
            char[][] mazeArray) {
        //loop will continue untill there is no more cordinates in the direct
        //path to print
        while (!rowStack.empty()) {
            //getting the "address" of the direct path out of the stacks
            int row = (int) rowStack.pop();
            int col = (int) colStack.pop();
            //putting the path into the array
            mazeArray[row][col] = '.';
        }//end of for loop that puts in the . for the solved path

        //this loop clears all 'w' from the array
        for (int row = 0; row < mazeArray.length; row++) {
            for (int col = 0; col < mazeArray[row].length; col++) {
                if (mazeArray[row][col] == 'w') {
                    mazeArray[row][col] = ' ';
                }

            }
        }//end of for loops to delete w
        return mazeArray;//the complete maze array will be returned
    }//end of print stack
    /**
     * This method can print out any 2d array. It is used to print the array
     * that contains the maze, it it used at the end once the maze is solved.
     * This method goes row by row printing every char in every col
     *
     * @param mazeArray the array that i want to print out
     */
    public static void printMaze(char[][] mazeArray) {
        for (int row = 0; row < mazeArray.length; row++) {
            for (int col = 0; col < mazeArray[row].length; col++) {
                System.out.print(mazeArray[row][col]);
            }//col for
            System.out.println();//to seperate rows
        }//row for
    }//end of print maze
}//end of class MazeSolver
```