18.20. *(Maze Traversal Using Recursive Backtracking) The grid of #s and dots (.) below is a two-dimensional array representation of a maze. The #s represent the walls of the maze, and the dots represent locations in the possible paths through the maze. A move can be made only to a location in the array that contains a dot.*

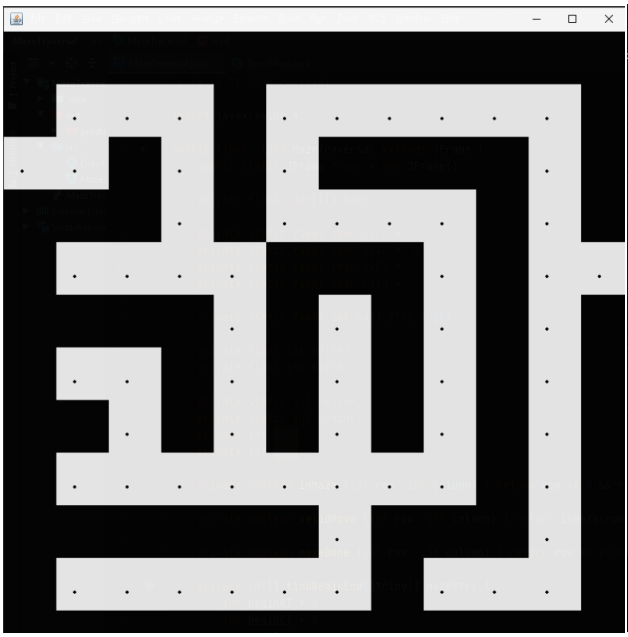
Write a recursive method (mazeTraversal) to walk through mazes like the one below. The method should receive as arguments a 12-by-12 character array representing the maze and the current location in the maze (the first time this method is called, the current location should be the entry point of the maze). As mazeTraversal attempts to locate the exit, it should place the character x in each square in the path. There's a simple algorithm for walking through a maze that guarantees finding the exit (assuming there's an exit). If there's no exit, you'll arrive at the starting location again. The algorithm is as follows: From the current location in the maze, try to move one space in any of the possible directions (down, right, up or left). If it's possible to move in at least one direction, call mazeTraversal recursively, passing the new spot on the maze as the current spot.

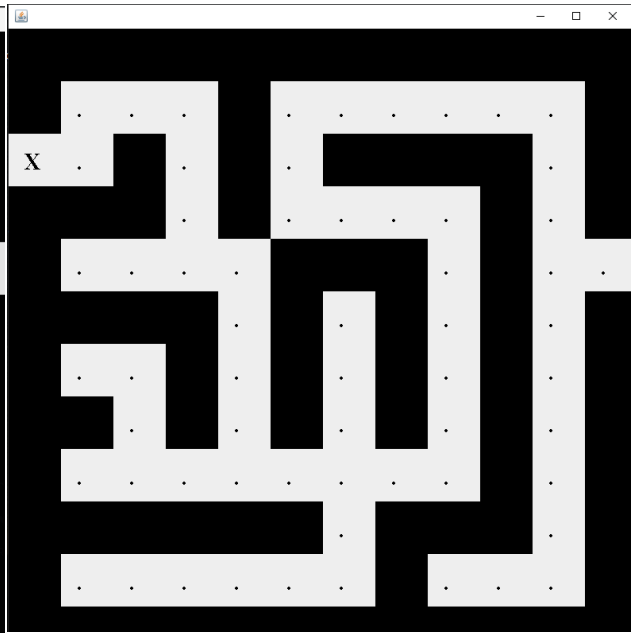
If it's not possible to go in any direction, "back up" to a previous location in the maze and try a new direction for that location (this is an example of recursive backtracking). Program the method to display the maze after each move so the user can watch as the maze is solved. The final output of the maze should display only the path needed to solve the maze—if going in a particular direction results in a dead end, the x's going in that direction should not be displayed. [Hint: To display only the final path, it may be helpful to mark off spots that result in a dead end with another character (such as '0').]

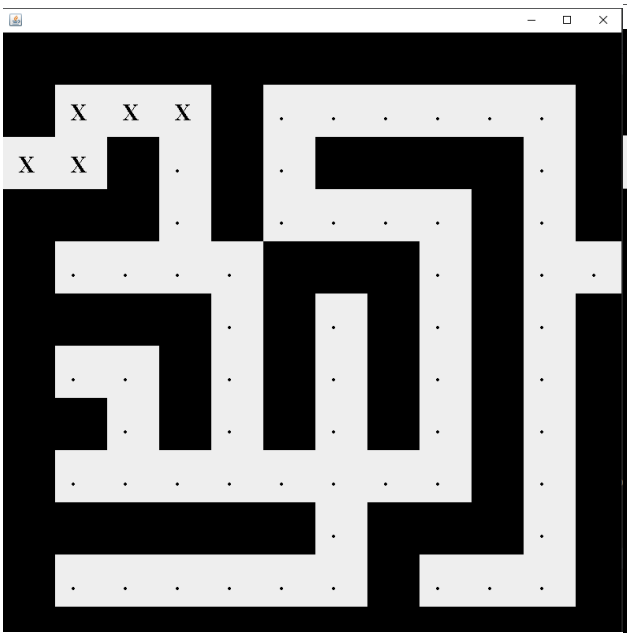
Example maze:

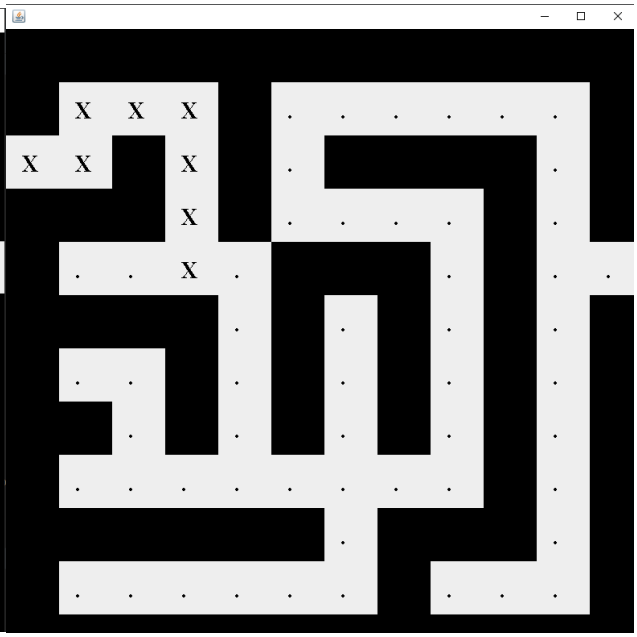
```
############
#...#......#
..#.#.#####.#
###.#.....#.#
#...#.###.#..
####.#.#.#.#.#
#..#.#.#.#.#.#
##.#.#.#.#.#.#
#.......#..#.#
######.####.#
#......#...#
############
```

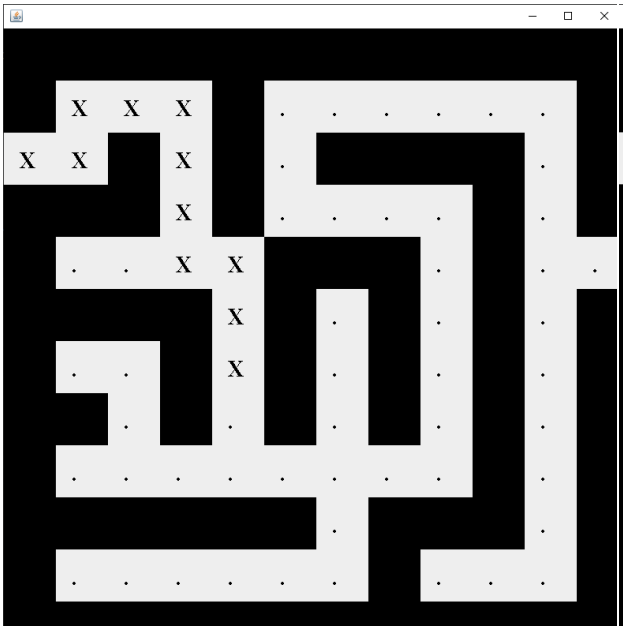
We created the desired program, complete with an animation to show the algorithm in the process of solving the maze. We tested it on the example maze given in the text:

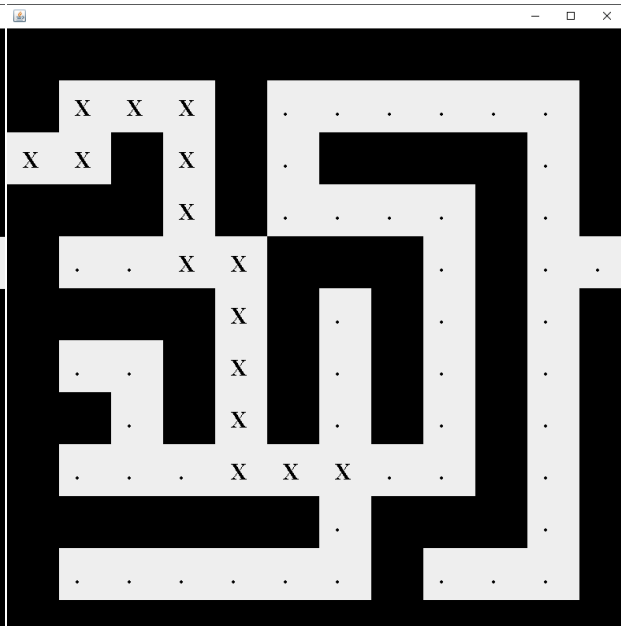


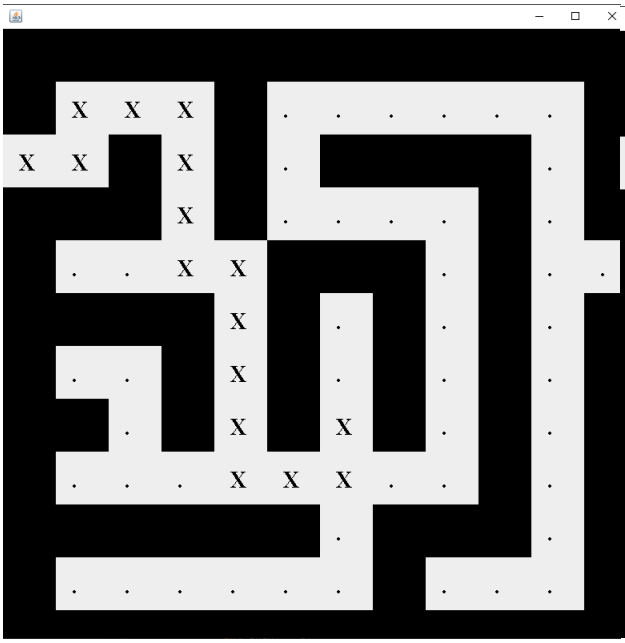


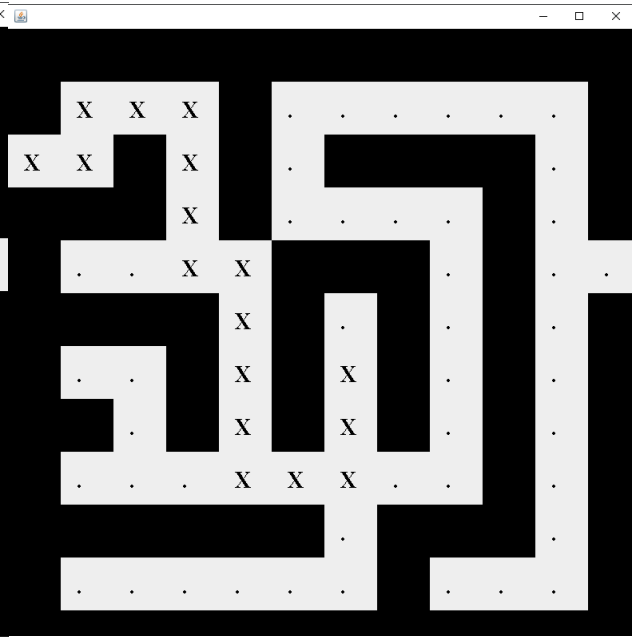


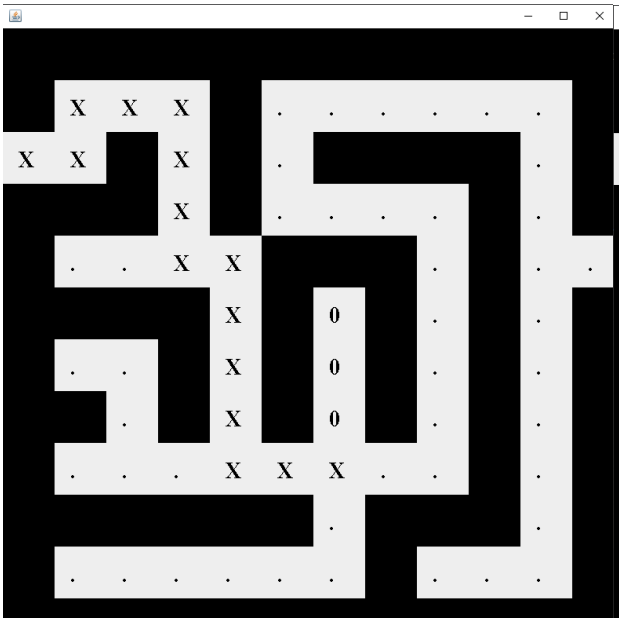


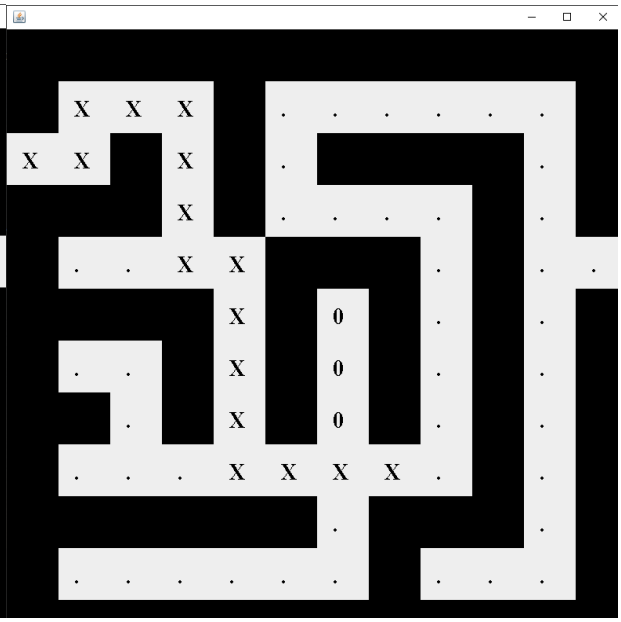


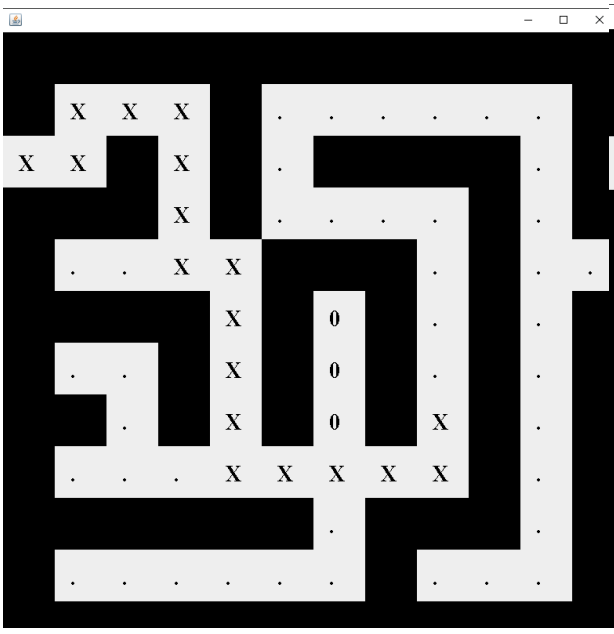


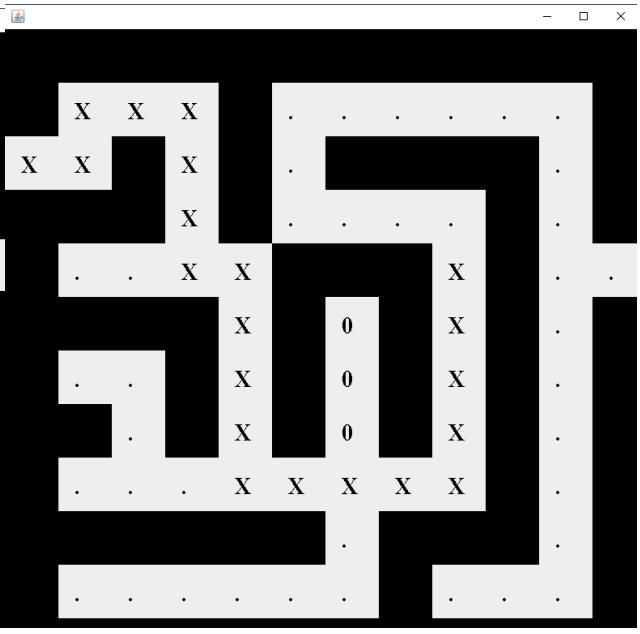


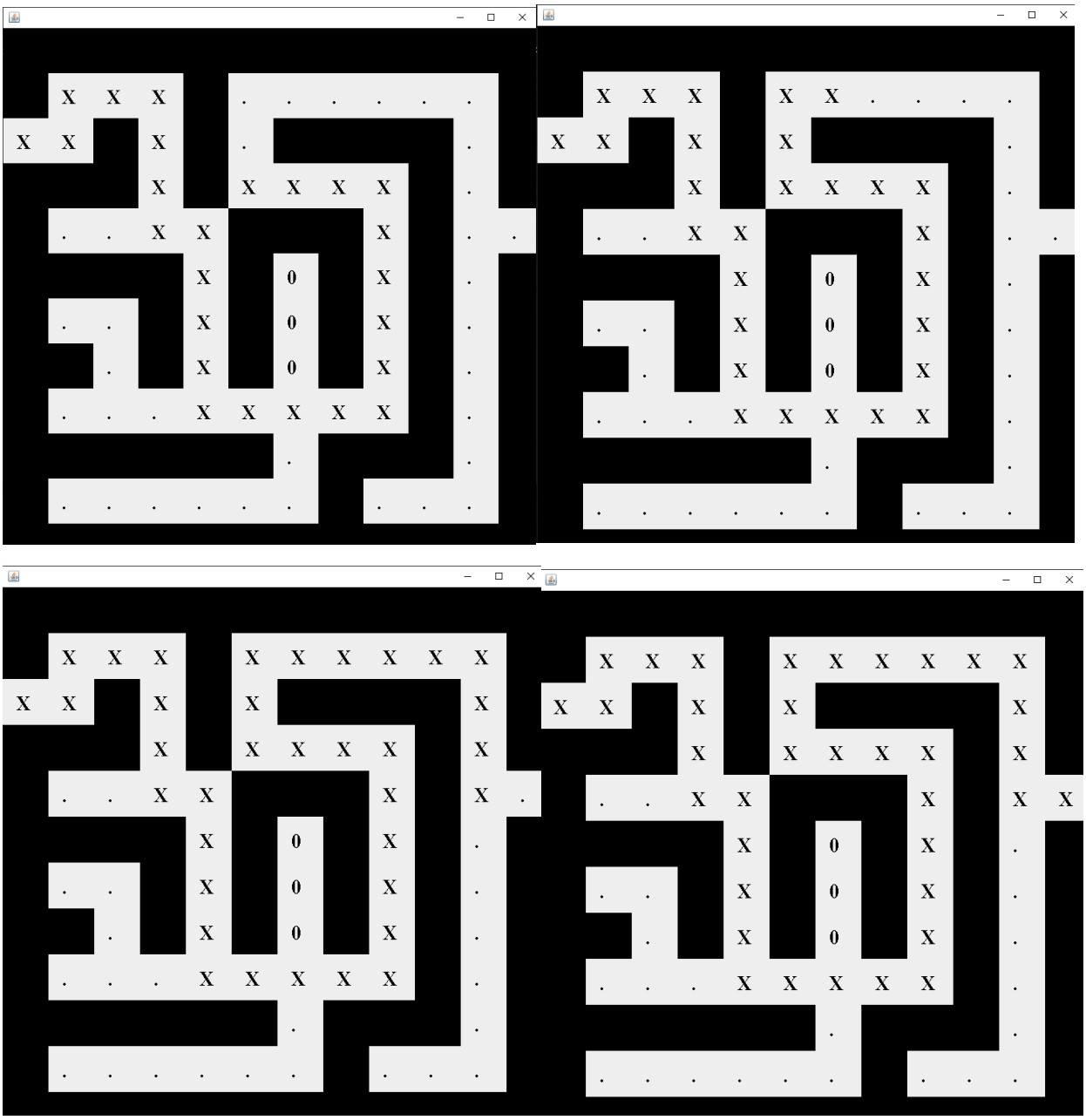




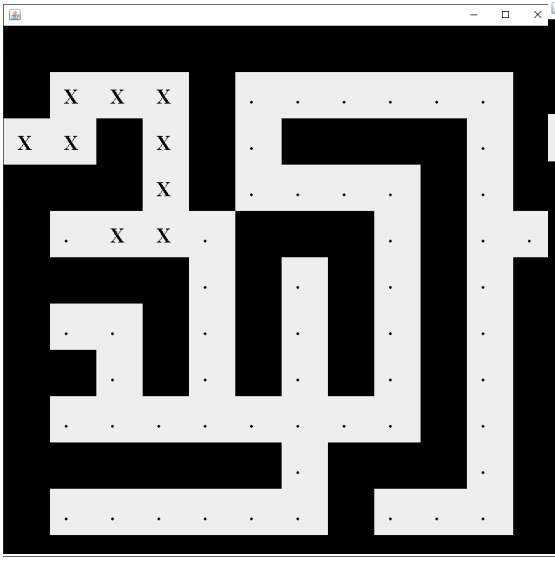


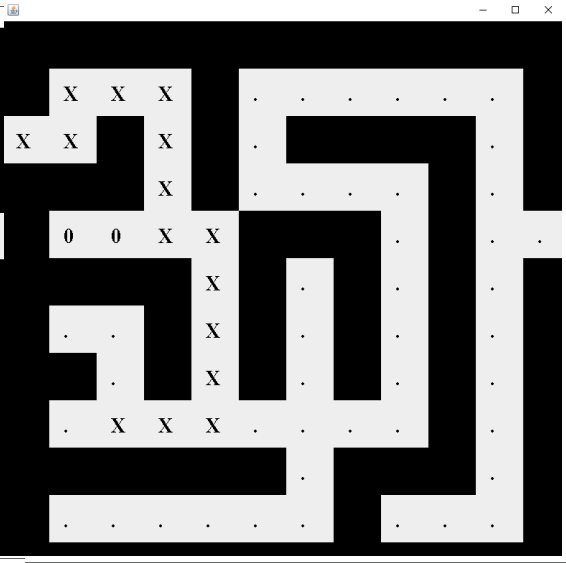


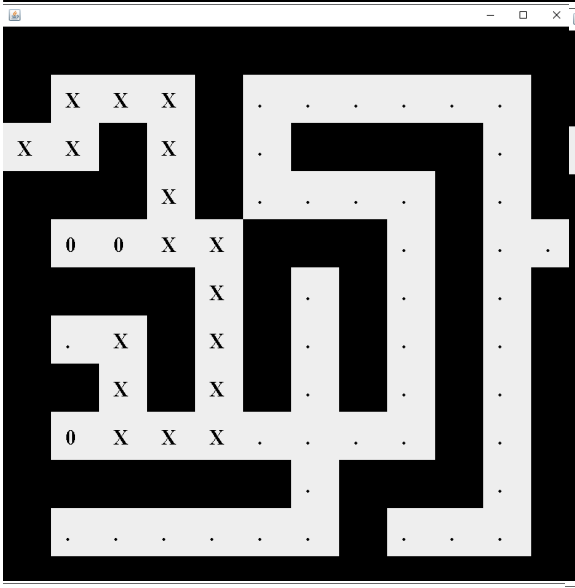


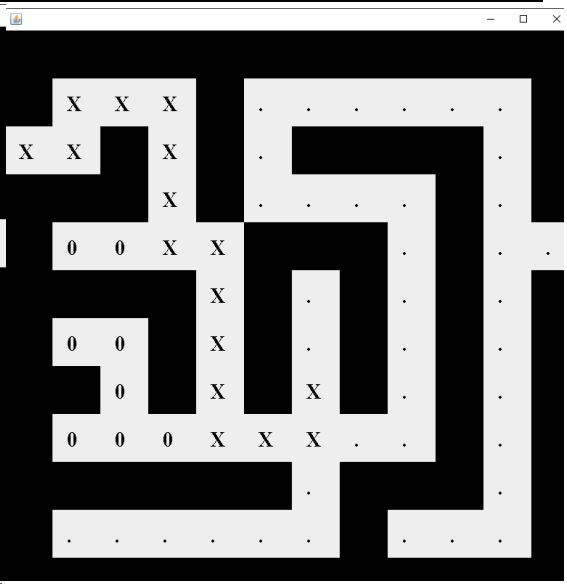


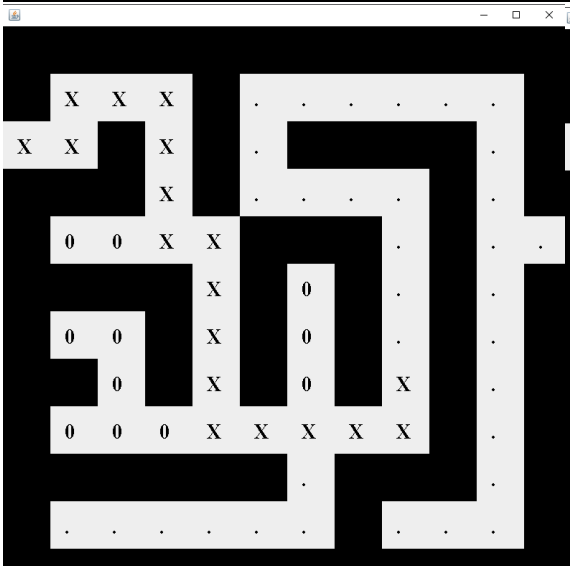
As the animation shows, the program marks with an X each square it travels to, until it hits a dead end, then marks it with a 0, backtracks to the last X, tries a different route from there, and if none is available, marks it with a 0, then backtracks to the last X, tries different routes etc. In this version of the program, the directions we try are up, right, down, and left, in that order, which leads to the result we see. If we change the order of directions, the algorithm follows a different path when trying to navigate the maze. For instance, left, up, right, down in order of preference leads to this:

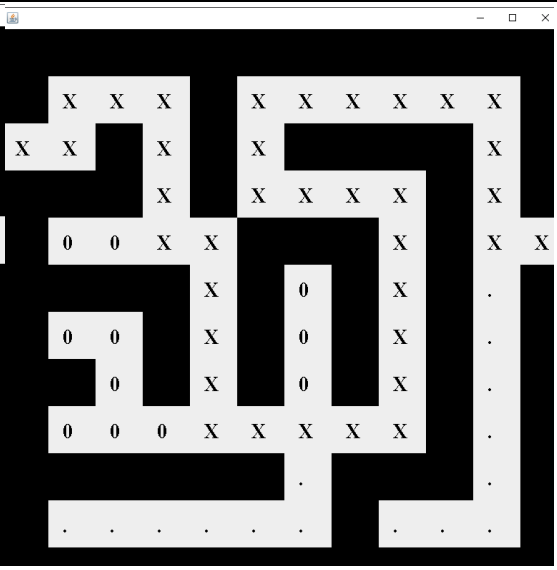




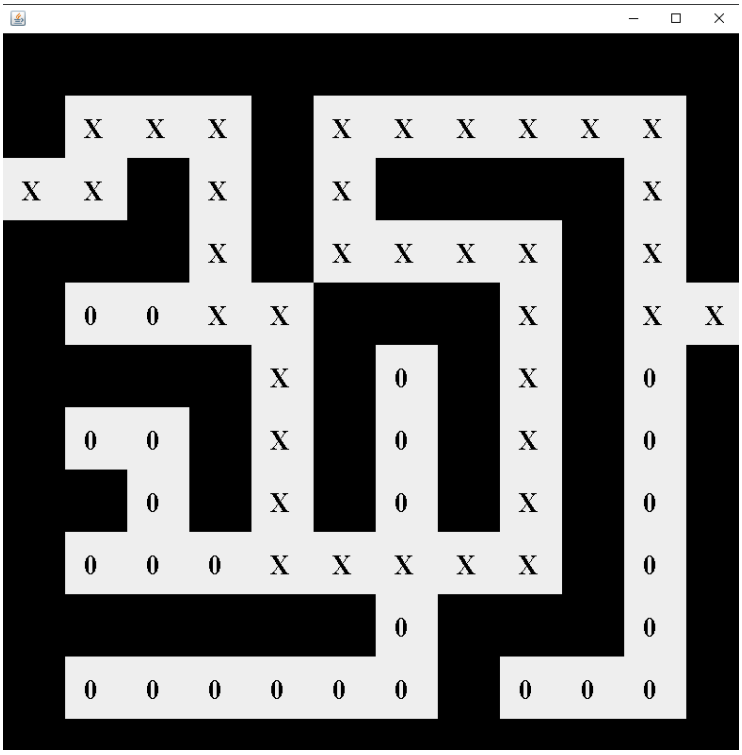




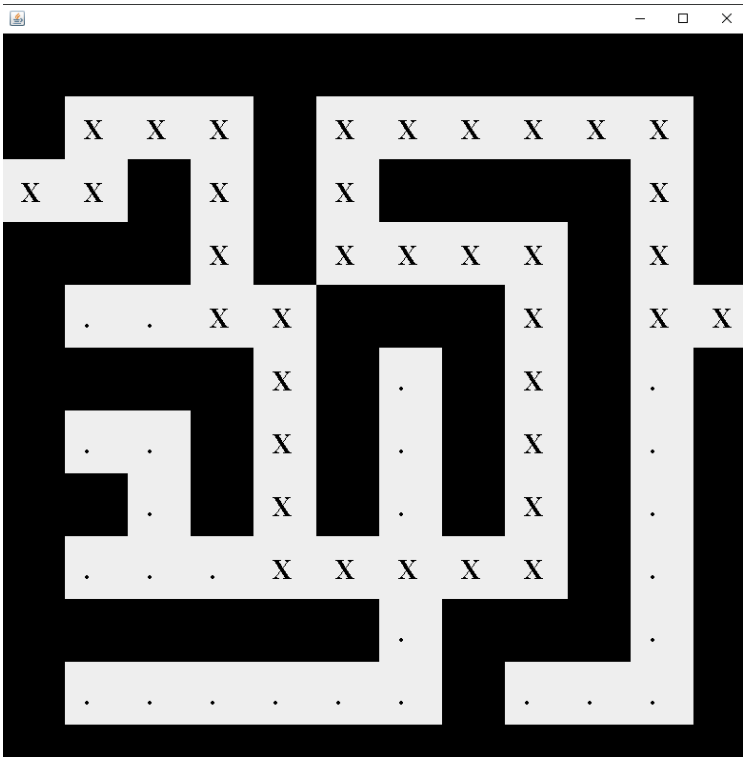




Just for fun we'll look at the worst order preference for this maze, which happens to be left, down, up, right, which leads us down every wrong path:



And the best order, which is any order preference that begins with right (e.g. right, left, up, down):



Now, after the animation completes, we print the initial maze and the solution to console (here is the output for the initial order configuration (up, right, down, left):

```
Maze:
#  #  #  #  #  #  #  #  #  #  #  #
#  .  .  .  #  .  .  .  .  .  .  #
.  .  #  .  #  .  #  #  #  #  .  #
#  #  #  .  #  .  .  .  .  #  .  #
#  .  .  .  .  #  #  #  .  #  .  .
#  #  #  #  .  #  .  #  .  #  .  #
#  .  .  #  .  #  .  #  .  #  .  #
#  #  .  #  .  #  .  #  .  #  .  #
#  .  .  .  .  .  .  .  .  #  .  #
#  #  #  #  #  #  .  #  #  #  .  #
#  .  .  .  .  .  .  #  .  .  .  #
#  #  #  #  #  #  #  #  #  #  #  #

Solved: true
#  #  #  #  #  #  #  #  #  #  #  #
#  X  X  X  #  X  X  X  X  X  X  #
X  X  #  X  #  X  #  #  #  #  X  #
#  #  #  X  #  X  X  X  X  #  X  #
#  .  .  X  X  #  #  #  X  #  X  X
#  #  #  #  X  #  0  #  X  #  .  #
#  .  .  #  X  #  0  #  X  #  .  #
#  #  .  #  X  #  0  #  X  #  .  #
#  .  .  .  X  X  X  X  X  #  .  #
#  #  #  #  #  #  .  #  #  #  .  #
#  .  .  .  .  .  .  #  .  .  .  #
#  #  #  #  #  #  #  #  #  #  #  #
```

Let us look at the code. It is rather long, so we will look at it by sections. Firs, MazeTraversal.java:

```
// Ex 18.20 MazeTraversal

import javax.swing.*;

public final class MazeTraversal extends JFrame {
    public static JFrame frame = new JFrame();

    private final char[][] maze;

    private static final char WALL = '#';
    private static final char DEAD = '0';
```

```
    private static final char OPEN = '.';
    private static final char PATH = 'X';

    private static final int WAIT_TIME = 250;

    private final int FRAME_HEIGHT = 800;
    private final int FRAME_WIDTH = 800;
    private final int HEIGHT;
    private final int WIDTH;

    private static int beginR;
    private static int beginC;
    private int endR;
    private int endC;
```

We begin by importing javax.swing.*, as we need to use the swing package for our animations. Our main class extends JFrame, once again because we will use JFrame for our animation. We begin by creating a JFrame to use, then we declare a number of variables we will use later: a 2D character array which will hold our maze, ints to represent the height and width of the maze, and ints to represent the starting and ending location of our maze (beginR, beginC, endR, endC). We also declare and initialize the characters we will use in our maze array: WALL='#', DEAD='0', OPEN='.', and PATH='X", as well as the WAIT_TIME, i.e. the time we wait between redraws in our animation. If you want the animations to go quicker, this could be set to 50 or even 10 for near-instantaneous animation. We also can adjust the FRAME_HEIGHT and FRAME_WIDTH we will eventually use in our JFrame here.

Now, the although the main method is at the end of the class, let us take a look at it next since it will guide us through the different parts of the program:

```
    public static void main(String[] args){
        frame.setSize(FRAME_HEIGHT,FRAME_HEIGHT);
        frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
        final String[] mazeStr = {
                "############",
                "#...#......#",
                "..#.#.####.#",
                "###.#....#.#",
                "#....###.#..",
                "####.#.#.#.#",
                "#..#.#.#.#.#",
                "##.#.#.#.#.#",
                "#........#.#",
                "######.###.#",
                "#......#...#",
                "############"};

        MazeTraversal m = new MazeTraversal(mazeStr);
```

First, we set the size of our JFrame to the FRAME_WEIGHT and FRAME_WIDTH we specified at the beginning of the class, here 800x800, then we setDefaultCloseOperation to EXIT_ON_CLOSE, i.e. the frame will terminate when we close the JFrame window. Then we specify a maze as a String array, since this is easier to type than a character array. Part of our

constructor for the class will parse this String array and turn it into a character array. Let us take a look at this constructor:

```
public MazeTraversal(String[] mazeStr) {
    this.WIDTH = mazeStr[0].length();
    this.HEIGHT = mazeStr.length;

    maze = new char[HEIGHT][WIDTH];
    for (int r = 0; r < HEIGHT; r++) {
        for (int c = 0; c < WIDTH; c++) {
             maze[r][c] = mazeStr[r].charAt(c);
        }
    }
```

First we take the number of length of each String in the array and assign it to the WIDTH dimension of our class, then take the number of Strings in our array to be the HEIGHT dimension of the class. Then create a 2D character array of the appropriate dimensions and we cycle through the string array and assign each character in the String array to the right position in the 2D character array. Finally in the constructor we have this:

```
    int[] locations = findBeginEnd(maze);

    this.beginR = locations[0];
    this.beginC = locations[1];
    this.endR = locations[2];
    this.endC = locations[3];
}
```

We then call our findBeginEnd method on the 2D character array maze we just made. This method finds the location of the beginning and end of the maze, and returns these locations in an array of length 4. Let us take a look at this method:

```
private int[] findBeginEnd(char[][] mazeC) {
    int beginRT = 0;
    int beginCT = 0;
    int endRT = 0;
    int endCT = 0;
    int numfound = 0;
    int[] locations = new int[4];
    // look at top row
    for(int column = 0; column < WIDTH; column++) {
        if (mazeC[0][column] == OPEN && numfound == 0) {
            beginRT = 0;
            beginCT = column;
            numfound++;
        } else if (mazeC[0][column] == OPEN && numfound == 1) {
            endRT = 0;
            endCT = column;
            numfound++;
        }
```

```
    }
    // look at bottom row
    for(int column = 0; column < WIDTH; column++) {
        if (mazeC[HEIGHT - 1][column] == OPEN && numfound == 0) {
            beginRT = HEIGHT - 1;
            beginCT = column;
            numfound++;
        } else if (mazeC[0][column] == OPEN && numfound == 1) {
            endRT = 0;
            endCT = column;
            numfound++;
        }
    }
    // look at left edge
    for(int row = 0; row < HEIGHT; row++) {
        if (mazeC[row][0] == OPEN && numfound == 0) {
            beginRT = row;
            beginCT = 0;
            numfound++;
        } else if (mazeC[row][0] == OPEN && numfound == 1) {
            endRT = row;
            endCT = 0;
            numfound++;
        }
    }
    // look at right edge
    for(int row = 0; row < HEIGHT; row++) {
        if (mazeC[row][WIDTH - 1] == OPEN && numfound == 0) {
            beginRT = row;
            beginCT = WIDTH - 1;
            numfound++;
        } else if (mazeC[row][WIDTH - 1] == OPEN && numfound == 1) {
            endRT = row;
            endCT = WIDTH - 1;
            numfound++;
        }
    }
    locations[0] = beginRT;
    locations[1] = beginCT;
    locations[2] = endRT;
    locations[3] = endCT;
    return locations;
}
```

Essentially, this program looks for dots (i.e. OPEN paths) along the edges of the maze. It first searches along the top edge, then the bottom edge, then the left edge, then the right edge. There is an int called numfound which keeps track of how many entries/exits we have already found. If we have not yet found any entries/exits, the first entry/exit we find will be labelled the beginning of the maze, and its row number will be assigned to beginRT, and column number assigned to begin CT. Once we find the first entry/exit, we increment numfound by 1 (so that numfound = 1), then the next entry/exit found will have its row number assigned to endRT and its column number assigned to endCT. These four values are placed in a array of length 4 called locations,

and this array is returned to the caller method, which, let us recall, is the MazeTraversal constructor:

```
    int[] locations = findBeginEnd(maze);

    this.beginR = locations[0];
    this.beginC = locations[1];
    this.endR = locations[2];
    this.endC = locations[3];
}
```

So the values from the locations array returned to the constructor are assigned to the appropriate class variables, beginR, beginC, endR, and endC. So, bird's eye view, the constructor has taken the String array maze, turned it into a 2D character array maze, found the height and width of the maze, and found the entry and exit. Let us go back to the main method that called the constructor:

```
MazeTraversal m = new MazeTraversal(mazeStr);
DrawMaze drawMaze = new DrawMaze(mazeStr, FRAME_HEIGHT, FRAME_WIDTH);
frame.add(drawMaze);
frame.setVisible(true);
```

So the MazeTraversal object created by the constructor is assigned the label m. We then make a DrawMaze object, feeding its constructor mazeStr, FRAME_HEIGHT, and FRAME_WIDTH as parameters. This object is then added to the JFrame frame and displayed. Ok, let's backtrack a little and look at DrawMaze.java:

```
// 18.20 DrawMaze.java
// Helps display maze

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

public class DrawMaze extends JPanel {

    private static final char WALL = '#';

    private int FRAME_HEIGHT;
    private int FRAME_WIDTH;
    private int dHEIGHT;
    private int dWIDTH;
    private int sWIDTH;
    private int sHEIGHT;
    private char[][] dmaze;

    public DrawMaze(char[][] maze, int frameHeight, int frameWidth) {
        this.dmaze = maze;
        this.FRAME_HEIGHT = frameHeight;
        this.FRAME_WIDTH = frameWidth;
        this.dWIDTH = maze[0].length;
        this.dHEIGHT = maze.length;
        sWIDTH = FRAME_WIDTH / dWIDTH;
        sHEIGHT = FRAME_HEIGHT / dHEIGHT;
```

```
    }

    public void setMaze(char[][] mazeS) {
        this.dmaze = mazeS;
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (int i = 0; i < dHEIGHT; i ++) {
            for (int j = 0; j < dWIDTH; j++) {
                if (dmaze[j][i] == WALL) {
                    g.setColor(Color.BLACK);
                    g.fillRect(sHEIGHT * i, sWIDTH * j, sHEIGHT, sWIDTH);
                }
                g.setColor(Color.BLACK);
                g.setFont(new Font("Serif", Font.BOLD, ((int) ((float)
sWIDTH/2.3))));
                g.drawString(Character.toString(dmaze[j][i]), (sHEIGHT/3) +
sHEIGHT * i, ((int) ((float) sWIDTH/1.5)) + sWIDTH * j);
            }
        }
    }
}
```

We import awt and swing to help us with graphics. DrawMaze extends JPanel, as we will draw our maze on a JPanel before adding it to our JFrame in the other class. We want to color WALL characters as black squares, so we set char WALL='#' to make it easier to refer to later. We then declare other variables useful for drawing the maze, such as frame width, frame height, the dimension of the maze (dHEIGHT and dWIDTH), the dimension of each square in the grid that we will use to draw the map (sWIDTH and sHEIGHT), and finally a 2D character array dmaze which will be updated to after every move through the maze and before every call to repaint so that the animation can display the current step in the algorithm.

Now, the constructor takes the 2D character maze and assigns it to its current solution state dmaze. Then it assigns to its own internal frame width and frame height variables the values it gets passed through the constructor. Using the 2D character array maze, it calculates the dimension of the maze, and then, by dividing the dimensions of the frame by the dimensions of the maze, figures out the dimensions of each square in the grid (sWIDTH and sHEIGHT).

Now, there is a setMaze method that will be used by MazeTraversal.java to update this class's internal dmaze to represent the current state of the solution algorithm.

Finally, the paintComponent method does most of the graphics work (we use an Override annotation to override JPanel's paintComponent method). As typical basic setup, it makes a graphics object as argument and calls the paintComponent from the parent class. Then we use nested loops to cycle through the maze and, for every square where we find a wall, paint a black rectangle. For every square, look at the corresponding character in the 2D character array and draw every String to its corresponding location on the grid (beforehand, we set the font to be Serif and Bold and about point size of square width divided by 2.3 – this last number was chosen

simply because it looked esthetically pleasing for a width range of different size mazes). When deciding where to place the beginning of each character in each square, we offset the character to 1/3 square height from the square's left edge and 1/3 square width from the square's bottom. Once again, these fractions were chosen so the character would be relatively centered for a large range of different maze sizes.

Now, back to our main class, once we've created a DrawMaze object and displayed the basic maze, we have the following line:

```
System.out.println("Maze: \n" + m);
```

This prints 'Maze:' and then calls MazeTraversal's toString method on m. Let us look at this method:

```
@Override
public String toString() {
    final StringBuilder result = new StringBuilder();
    for (int row = 0; row < HEIGHT; row++) {
        for (int column = 0; column < WIDTH; column++)
            result.append(maze[row][column] + (column == WIDTH - 1 ? "" : "
"));
        result.append(row == HEIGHT - 1 ? "" : "\n");
    }
    return result.toString();
}
```

We must use the @Override annotation to override JFrame's toString method. We then create a StringBuilder object to build our maze string we want to display. Then, using nested for loops, we go through each row, and for each row, we add the character in the corresponding 2D character array to our StringBuilder object, followed by a space, unless we are at the end of a row, in which case we add nothing. Then at the end of each row, we append a new line marker (\n) unless we are on the last row, in which case we add nothing. As the end, we take our StringBuilder object (result) and call StringBuilder's toString method, which returns a String object corresponding to the StringBuilder, and we pass that String out of MazeTraversal's toString method and back to the print state. Let us recall, the result looks something like this:

```
Maze:
#  #  #  #  #  #  #  #  #  #  #  #
#  .  .  .  #  .  .  .  .  .  .  #
.  .  #  .  #  .  #  #  #  #  .  #
#  #  #  .  #  .  .  .  .  #  .  #
#  .  .  .  .  #  #  #  .  #  .  .
#  #  #  #  .  #  .  #  .  #  .  #
#  .  .  #  .  #  .  #  .  #  .  #
#  #  .  #  .  #  .  #  .  #  .  #
#  .  .  .  .  .  .  .  .  #  .  #
#  #  #  #  #  #  .  #  #  #  .  #
#  .  .  .  .  .  .  #  .  .  .  #
#  #  #  #  #  #  #  #  #  #  #  #
```

Now, back to our main method, the next line is the following:

```
System.out.println("\nSolved: "+ m.mazeTraversal(beginR, beginC, drawMaze));
```

First, let us look at the mazeTraversal method call inside the print statement. We pass the beginning square to the method and the drawMaze object to the method. Before we look at the mazeTraversal method, let us look at some helper methods that it will call during its execution:

```
private boolean inMaze (int row, int column) {
    return row >= 0 && row < HEIGHT && column >= 0 && column < WIDTH;
}

private boolean validMove (int row, int column) {
    return inMaze(row, column) && maze[row][column] == OPEN;
}

private boolean mazeDone (int row, int column) {
    return row == endR && column == endC;
}
```

The function of these methods is rather clear from their names. The inMaze method takes a set of row and column coordinates and returns whether of not those coordinates are actually inside the maze grid (i.e. not out of bounds). Next, the validMove method checks to determine is a possible move is value. For a move to be valid, it must be inside the maze (inMaze method) and it cannot be a wall, a dead path, or a square on our current path. Lastly, the mazeDone method checks to see whether a given square is the end square (i.e. row equal to endR and column equal to endC). Lastly, one last helper method which pauses the animation for a predetermined number of miliseconds:

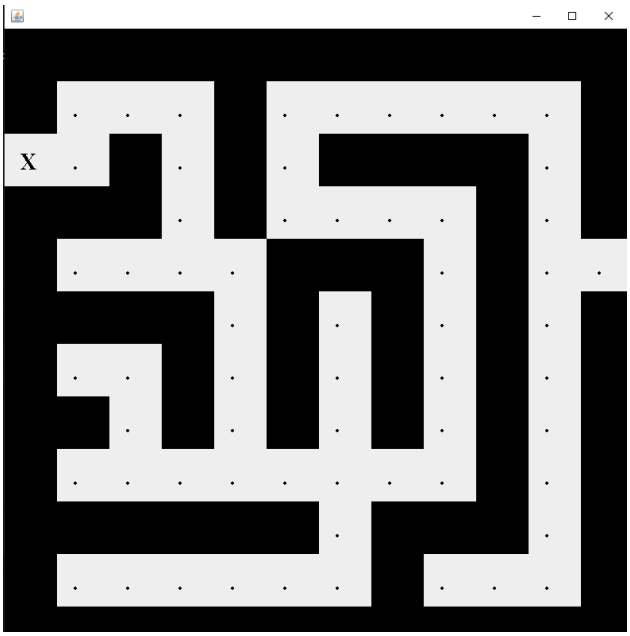
```
public static void wait(int ms){
    try {
        Thread.sleep(ms);
    }
    catch(InterruptedException ex) {
        Thread.currentThread().interrupt();
    }
}
```

Now finally to the mazeTraversal method:

```
public boolean mazeTraversal(int row, int column, DrawMaze dMaze) {
    if (!validMove(row,column))
        return false;

    dMaze.setMaze(maze);
    dMaze.repaint();
    wait(WAIT_TIME);
    maze[row][column] = PATH;


    if (mazeDone(row,column))
        return true;
```

```
    if (mazeTraversal(row - 1, column, dMaze)) // up
        return true;
    if (mazeTraversal(row, column + 1, dMaze)) // right
        return true;
    if (mazeTraversal(row + 1, column, dMaze)) // down
        return true;
    if (mazeTraversal(row, column - 1, dMaze)) // left
        return true;

    maze[row][column] = DEAD;
    return false;
}
```

Initially, we start on beginR and beginC, which is a valid move, so the method updates dMaze's internal maze and repaints the animation, then waits, then labels the beginning square with an X.

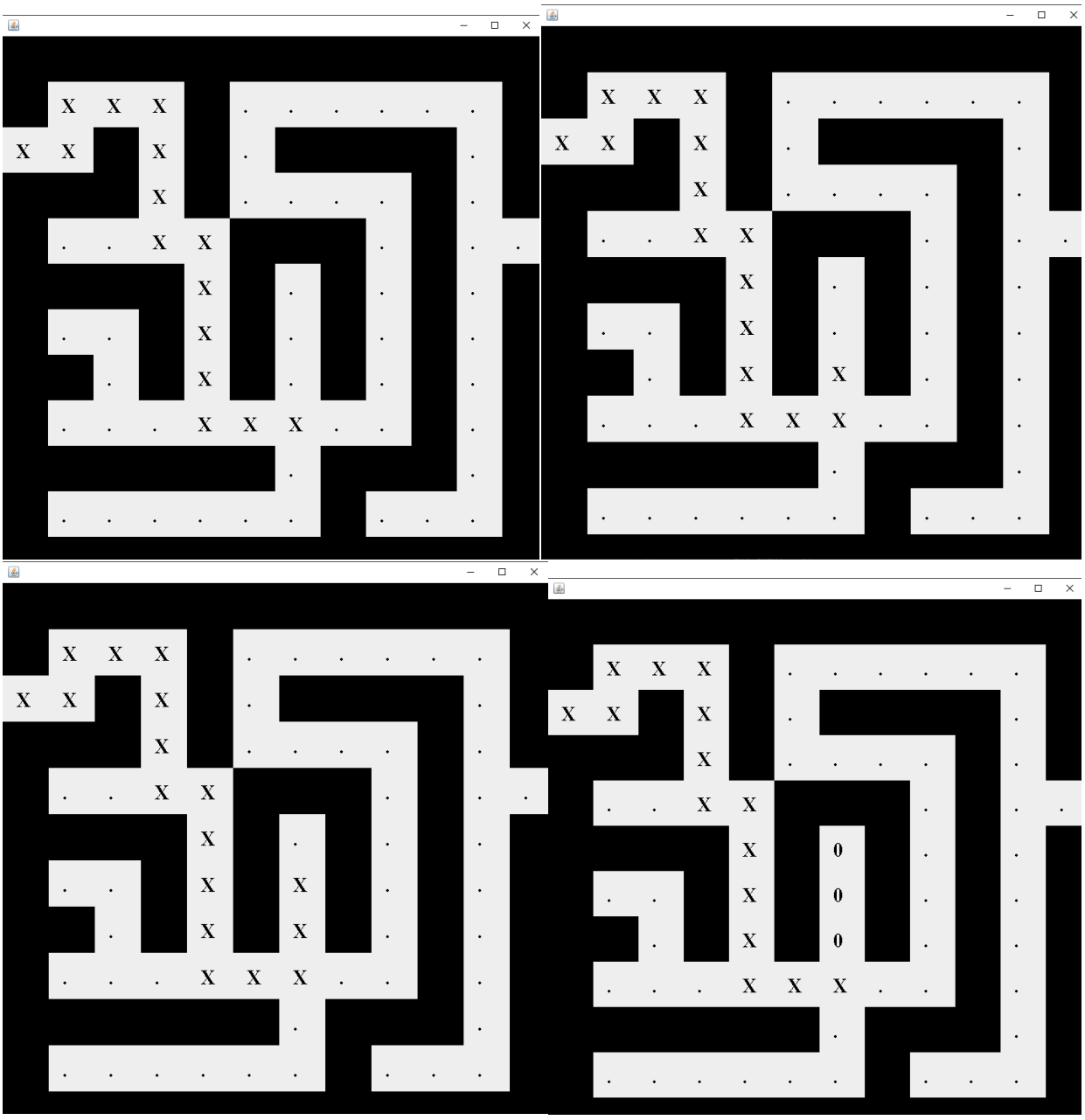


If we are not done (which we are not), we begin cycling through directions looking for the next square to move to. First we recursively call mazeTraversal on the square above our current square. This square is not a valid move, so the recursive call returns false, and the caller mazeTraversal we skip that if branch and move to the next one, which recursively calls mazeTraversal on the square to the right of our current square. This square is valid, so we update our DrawMaze object's internal maze and repaint, then wait a moment, then label the new square with an X. Then, inside this recursive call, we start cycling through different recursive method calls for each direction. The recursive calls build a chain like this:

mazeTraversal(2,0) → mazeTraversal(2,1) → mazeTraversal(1,1)
mazeTraversal(2,0) → mazeTraversal(2,1) → mazeTraversal(1,1) → mazeTraversal(0,1) FALSE
mazeTraversal(2,0) → … → .. → mazeTraversal(1,2) → mazeTraversal(0,2) FALSE
mazeTraversal(2,0) → … → .. → … → mazeTraversal(1,3) → mazeTraversal(0,3) FALSE
mazeTraversal(2,0) → … → .. → … → mazeTraversal(1,3) → mazeTraversal(1,4) FALSE

mazeTraversal(2,0) → … → .. → … → … → mazeTraversal(2,3) → mazeTraversal(1,3) F
mazeTraversal(2,0) → … → .. → … → … → mazeTraversal(2,3) → mazeTraversal(2,4) F
mazeTraversal(2,0) → … → .. → … → … → mazeTraversal(2,3) → mazeTraversal(3,3) → etc

And you keep building provision paths that may need to be backtracked. An example of that is what happens here:



In the upper left image, we check up, it is valid, so we move up and mark it with an X. Then in the upper right image, we check up, it is valid, so we move up and mark it with an X. Then in the bottom left image, we check up, it is valid, so we move up and mark it with an X. Now, we are at the last square in this dead end. We check up, there is a wall, we check right, there is a wall, we

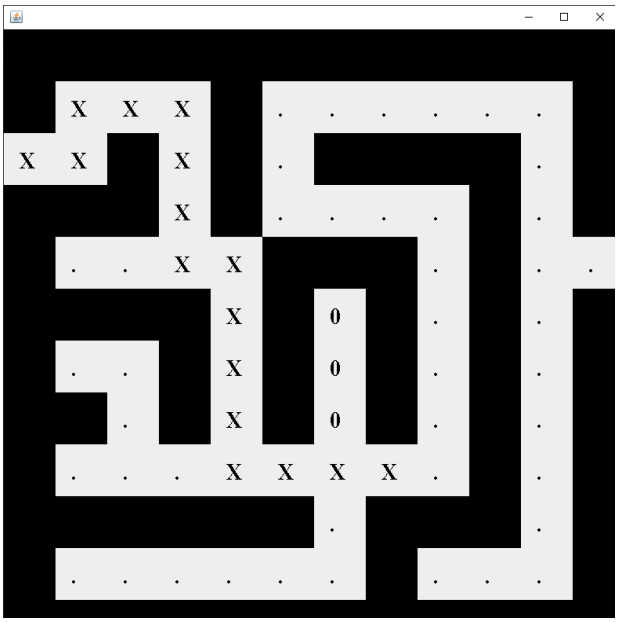
check down, already travelled path, and check left, there is a wall. At this point we arrive at this line:

```
    maze[row][column] = DEAD;
    return false;
}
```

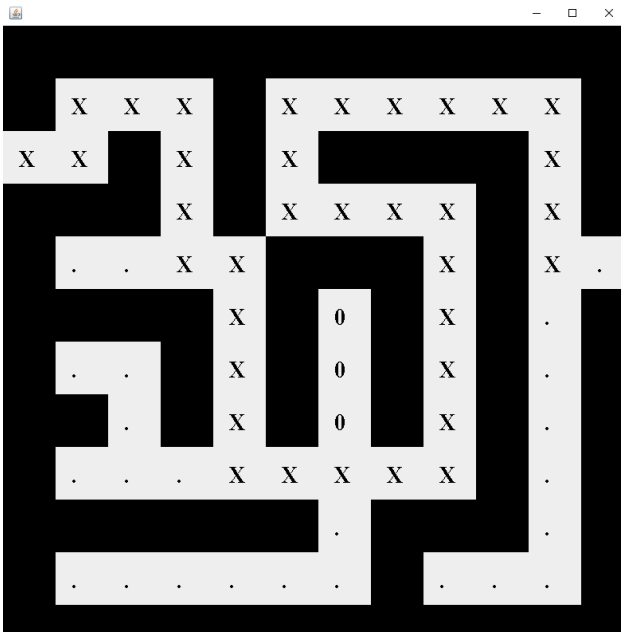
We mark this square with a 0 (DEAD) and return false. This false goes to the square that called mazeTraversal on the dead-end square (i.e. the square under the dead-end square). This square now calls mazeTraversal on the square to the right, which returns false since it is a wall, then calls mazeTraversal on the square below it, which returns false since it is already marked with an X, then on the square to the left, which returns false (WALL). At this point, the square below the dead-end square is marked as DEAD, and false is returns to the square that called mazeTraversal on this square (the square below it). Siimilarly, this square cycles through the other directions and finds WALL, PATH, WALL, and so it reached the end of the method and is marked as DEAD and returns false. Now we are back at the square we were at in the upper left image. At this point the

```
if (mazeTraversal(row - 1, column, dMaze))
```

branch is not taken since the recursive method returns false, so we proceed to the next direction (right), which is a valid move, so gets marked with an X and we try continuing in that direction:



On a macro scale, an dead-end routes will end up returning false on all the recursive mazeTraversal method calls, which will cause the algorithm to backtrack to the last valid square and continue from there trying a new direction. Eventually, we are just before the end square, e.g. something like this:



We call mazeTraversal recursive on the square above our current square, and it is a PATH, so this recursive call returns false. Then we call mazeTraversal on the square to the right of our current square, which is valid, so we pass the maze to our DrawMaze dMaze, redraw, wait, and mark the final square with PATH. This time, when we get to this if statement:

```
if (mazeDone(row,column))
    return true;
```

The condition is true, so we return true. This true return will be passed through every recursive method call that got us to this point until the true gets passed back to our original mazeTraversal call in the main method (mazeTraversal(beginR, beginC, dMaze)). Looking back to the original call in our main method…

```
        System.out.println("\nSolved: "+ m.mazeTraversal(beginR, beginC,
drawMaze));
        drawMaze.repaint();
        System.out.println(m);
    }
}
```

We print “Solved: true”, then repaint the drawMaze to display the last square marked as X. Lastly, we call toString on MazeTraversal object m and print our solution to console.