```
/*
 * Implemented as a two-dimensional character array in which passages are marked
 * with 0s, walls by 1s, exit position by the letter e and the initial position
 * by the letter b. We will use recursion with backtracking to write a program
 * that asks the user to enter the dimensions of a maze and the maze, finds and
 * prints out the graphical solutions (optional) of all possible paths( a
 * solution is printed for each path, each maze entered may have multiple
 * solutions) by replacing the 0s with ps in each path.
 */
package maze;

import java.util.Arrays;

/**
 * 2018-10-14
 * @author James Garringer
 */
public class Maze {
   private int x;
   private int y;
   private char [][] maze;
   private int[] start = new int[2];
   private int[] end = new int[2];
   private int count = 1;

   public Maze()
   {
      x = 5;
      y = 5;
```

```
    maze = new char[y][x];
    for(int i = 0; i < y; i++)
    {
        for(int j = 0; j < x; j++)
        {
            maze[i][j] = '0';
        }
    }
    capper(0);
    maze[1][0] = 'b';
    maze[1][2] = '1';
    maze[1][4] = '1';
    maze[2][0] = '1';
    maze[2][2] = '1';
    maze[2][4] = 'e';
    maze[3][4] = '1';
    capper(y - 1);
    findStart();

}

/**
 * Creates an object of Maze
 * @param width The x dimension of the maze
 * @param height The y dimension of the maze
 * @param input  The actual maze itself
 */
public Maze(int width, int height, char[][] input)
{
```

```
    x = width;
    y = height;
    maze = Arrays.copyOf(input, input.length);
    findStart();
}

/**
 * This constructor sets the width and the height of the maze, and then
 * creates the maze based on a string passed to it.
 * @param width The overall width of the maze.
 * @param height The overall height of the maze.
 * @param str The string of the maze. Listed as one continuous string of 1s,
 * 0s, b, and e.
 */
public Maze(int width, int height, String str)
{

    x = width;
    y = height;
    maze = new char[y][x];
    for(int i = 0; i < y ; i++)
    {
        for(int j = 0; j < x; j++)
        {

            maze[i][j] = str.charAt(j + (i * x));

        }
    }
```

```
    findStart();
}

/**
 * In the case of no parameters passed.
 */
public void solve()
{
    solve(startX(), startY());
}

/**
 * Designed to first check if the current position is the end of the maze.
 * If it is the end this method prints out the maze's solution.
 * Next if the current location is not the beginning then it will change
 * the character held at this location to a 'p' to indicate the path of
 * traversal.
 * Then we check if positions are available to move to in a clockwise manner
 * starting from the east of the current position. Also checks the boundary
 * of the maze is not being crossed. If the position is available the process
 * begins anew using the coordinates of the available position.
 *
 * @param x X coordinate of the position being considered
 * @param y Y coordinate of the position being considered
 */
public void solve(int x, int y)
{
    //base case
    if (maze[y][x] == 'e')
```

```
{
   System.out.println("Maze number" + count);
   for(int i = 0; i < this.y; i++)
   {
      for(int j = 0; j < this.x; j++)
      {
         System.out.print(maze[i][j]);
      }
      System.out.println("");
   }

   count++;
}

   if (maze[y][x] != 'b' && maze[y][x] != 'e'){maze[y][x] = 'p';}

   //Check if position east of current is available
   if (checkAvailable(x + 1, y))
   {
      solve(x + 1, y);
      if (maze[y][x + 1] != 'e')
         maze[y][x + 1] = '0';
   }
   //Check if position south of current is available
   if (checkAvailable(x, y + 1))
   {
      solve(x , y+1);
      if (maze[y + 1][x] != 'e')
         maze[y + 1][x] = '0';
```

```
        }
        //Check if position west of current is available
        if(checkAvailable(x - 1, y))
        {
            solve(x - 1 , y);
            if (maze[y][x - 1] != 'e')
                maze[y][x - 1] = '0';
        }
        //Check if position north of current is available
        if (checkAvailable(x, y -1))
        {
            solve(x , y - 1);
            if (maze[y - 1][x] != 'e')
                maze[y - 1][x] = '0';
        }

}
/**
 * Checks if the position indicated by the arguments passed into the parameters
 * is an available location to move to.
 * @param x x coordinate of the position
 * @param y y coordinate of the position
 * @return True if the position is available, false otherwise.
 */
private boolean checkAvailable(int x, int y)
{
    if (x >= 0 && x <= this.x && y >= 0 && y <= this.y)
        return maze[y][x] == '0' || maze[y][x] == 'e';
    return false;
```

```
}

/**
 * Simple way to get a wall of 1s
 * @param row Which row is all 1s
 */
private void capper(int row)
{
   for(int i = 0; i < x; i++)
   {
      maze[row][i] = '1';
   }
}

/**
 * Helps to determine the array x and y of where the start of the maze is
 */
public void findStart()
{
   for (int i = 0; i < y; i++)
   {
      for(int j = 0; j < x; j++)
      {
         if (maze[i][j] == ('b')){start[0] = i;start[1] = j;}
      }
   }
}

/**
```

```
 * Helps locate the end of the maze
 */
public void findEnd()
{
   for (int i = 0; i < y; i++)
   {
      for(int j = 0; j < x; j++)
      {
         if (maze[i][j] == ('e')){end[0] = i;end[1] = j;}
      }
   }
}

/**
 * Getter for the x position of the start
 * @return x coordinate of the start
 */
public int startX()
{
   return start[1];
}

/**
 * Getter for the y position of the start
 * @return Y coordinate of the start
 */
public int startY()
{
   return start[0];
```

```
    }
    /**
     * Exists to test the constructor to make sure that it was reading in the
     * string properly
     */
    public void print()
    {
        for(int i = 0; i < y; i++)
        {
            for (int j = 0; j < x; j++)
            {
                System.out.print(maze[i][j]);
            }
            System.out.println("");
        }
    }

}
```