```
/*
Backtracking algorithms try each possibility until they find the right
one.
It is a depth-first search of the set of possible solutions.
During the search, if an alternative doesn't work, the search backtracks
to
the place which presented different alternatives, and tries the next
alternative.
When the alternatives are exhausted, the search returns to the previous
choice point
and tries the next alternative there.
If there are no more choice points, the search fails.
This is usually achieved in a recursive function.
Backtracking is similar to a depth-first search but uses even less space,
keeping just one current solution state and updating it.
*/

//********************************************************************
//  Maze.java       Author: Lewis/Chase
//
//  Represents a maze of characters. The goal is to get from the
//  top left corner to the bottom right, following a path of 1's.
//********************************************************************

// Adapted for NetBeans by Andrés Gómez de Silva Garza, 18/11/09

package maze;

/**
 *
 * @author AGOMEZDG
 */
public class Main {
      private final int CLEAR = 1;
      public final int PATH = 2;
      private int[][] grid;

      public Main(int[][] grid) {
            this.grid = grid;
      }


      /**************************************************************
     Attempts to recursively traverse the maze.
     Inserts a special character (PATH) indicating locations
     that eventually become part of the solution.
      **************************************************************/
      public boolean tryNextMove(int row, int column) {
            if ( !valid (row, column) )
                  return false; // impossible path
            else if ( solved(row, column) ) {
            setPath(row, column);
            return true;  // the maze is solved
        }
        else {
            setPath(row, column);   // mark
```

```
            boolean done = tryNextMove (row+1, column);     // down
            if (!done)
                  done = tryNextMove (row, column+1);  // right
            if (!done)
                  done = tryNextMove (row-1, column);  // up
            if (!done)
                  done = tryNextMove (row, column-1);  // left
            if (!done)
                  removePath(row, column); // backtrack
            return done;
            }
      }

      // Registra el orden del recorrido (inicia en n=PATH).
      // El método tiene un solo return y simplifica el if-else inicial
      // Trata de moverte a la posición (row, column)
      public boolean tryNextMove(int row, int column, int n) {
            boolean done = false;

            if ( valid (row, column) ) {
                  setPath(row, column, n);
            if ( solved(row, column) )
                  done = true;
            else {
                  done = tryNextMove(row+1, column, n+1);
                  if (!done)
                        done = tryNextMove(row, column+1, n+1);
                  if (!done)
                        done = tryNextMove(row-1, column, n+1);
                  if (!done)
                        done = tryNextMove(row, column-1, n+1);
                  if (!done)
                        removePath(row, column);
            }
            }

            return done;
      }

      /* tryNextMove unmarks each square as it retreats while
backtracking
       * (removePath). Removing the marks from the blind alleys that the
       * program tries along the way ensures that the final path will be
       * displayed correctly. However, if the goal is to find the finish
       * square in the shortest possible time, it is more efficient to
leave
       * these markers in place (Lewis & Chase solution)
       */


      /**************************************************************
     Determines if a specific location is valid.
      **************************************************************/
      private boolean valid(int row, int column) {
            boolean result = false;
            /** check if cell is in the bounds of the matrix */
            if (row >= 0 && row < grid.length &&
```

```
      column >= 0 && column < grid[row].length)
      /**  check if cell is not blocked */
      if (grid[row][column] == CLEAR)
            result = true;
      return result;
 }

 private boolean solved(int row, int column) {
      return row==grid.length-1 && column==grid[0].length-1;
 }

 private void setPath(int row, int column) {
      grid[row][column] = PATH;
 }

 private void setPath(int row, int column, int i) {
      grid[row][column] = i;
 }

 private void removePath(int row, int column) {
      grid[row][column] = CLEAR;
 }

 /********************************************************************
Returns the maze as a string.
 ********************************************************************/
 public String toString() {
      StringBuilder result = new StringBuilder("\n");
      for (int row=0; row < grid.length; row++) {
      for (int column=0; column < grid[row].length; column++) {
            result.append(grid[row][column]);
            if (grid[row][column] < 10)
                  result.append("   ");
            else if (grid[row][column] < 100)
                  result.append("  ");
      }
      result.append("\n");
      }
      return result.toString();
 }

 public static void main(String[] args) {

      // 1 indicates a clear path
      // 0 indicates a blocked path
      int[][] grid = {{1,1,1,0,1,1,0,0,0,1,1,1,1},
                  {1,0,1,1,1,0,1,1,1,1,0,0,1},
                  {0,0,0,0,1,0,1,0,1,0,1,0,0},
                  {1,1,1,0,1,1,1,0,1,0,1,1,1},
                  {1,0,1,0,0,0,0,1,1,1,0,0,1},
                  {1,0,1,1,1,1,1,1,1,0,1,1,1,1},
                  {1,0,0,0,0,0,0,0,0,0,0,0,0},
                  {1,1,1,1,1,1,1,1,1,1,1,1,1} };

      Main labyrinth = new Main(grid);

      System.out.println(labyrinth);
```

```
            if (labyrinth.tryNextMove(0, 0, labyrinth.PATH)) // posición
inicial válida
            //if (labyrinth.tryNextMove(0, 0))
            System.out.println("The maze was successfully traversed!");
            else
            System.out.println("There is no possible path.");

            System.out.println(labyrinth);
   }
}
```