CS2030 Lab #2: (n^2 - 1) Puzzle

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

The `(n`^{2}` - 1)` Puzzle

Topic Coverage

- Abstraction and Encapsulation
- Immutable List
- CS2030 Java Style Guide

Problem Description

The 15-Puzzle is a classic sliding puzzle game consisting of a 4-by-4 grid of 15 numbered tiles all jumbled up. The following is an example.

```
  7  15  12   4
 11   9   8   2
  .   1  10   3
 13   5   6  14
```

The objective of the game is to un-jumble the puzzle by repeatedly sliding the tiles to the empty space (denoted by `.`) to attain the solution below.

```
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15   .
```

We shall generalize the 15-Puzzle game to a `(n`^{2}` - 1)` puzzle.

The Task

The task is to let our user play the game (rather than solve the game which is a non-trivial task). You will first write a `Grid2D` class to represent a two-dimensional grid. Next, write the `Puzzle` class that makes use of `Grid2D` to move the tiles and determine if the puzzle is solved.

This task is divided into several levels. You may submit as soon as you complete one level. There is no restriction on the number of attempts.

You will need to complete ALL levels to get full credit for this lab.

Level 1

By making use of the `ImList` class provided, create a `Grid2D` class to represent a variable-sized two-dimensional grid containing integer values with the following specifications:

- a constructor `Grid2D(List<Integer> list, int numOfCols)` that takes in a `List` of values and the number of columns `numOfCols`. You may assume that `numOfCols > 0`.
- a `toString()` method that outputs the values stored in the grid such that values within each row are separated by commas, with each row separated by semi-colons.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> new Grid2D(List.of(1, 2, 3), 1)
$.. ==> {1;2;3}

jshell> new Grid2D(List.of(1, 2, 3), 2)
$.. ==> {1,2;3}

jshell> new Grid2D(List.of(1, 2, 3), 3)
$.. ==> {1,2,3}

jshell> new Grid2D(List.of(1, 2, 3), 4)
$.. ==> {1,2,3}
```

Level 2

Define an overloaded constructor `Grid2D(int numOfCols)` that takes in the number of columns `numOfCols` and creates an empty two-dimensional grid.

Include the `add(int elem)` method that adds to the current two-dimensional grid.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> Grid2D emptyGrid = new Grid2D(2)
emptyGrid ==> {}

jshell> emptyGrid.add(1)
$.. ==> {1}

jshell> emptyGrid.add(1).add(2)
$.. ==> {1,2}

jshell> emptyGrid.add(1).add(2).add(3)
$.. ==> {1,2;3}

jshell> emptyGrid
emptyGrid ==> {}

jshell> new Grid2D(List.of(1,2,3), 2)
$.. ==> {1,2;3}

jshell> new Grid2D(List.of(1,2,3), 2).add(4)
$.. ==> {1,2;3,4}

jshell> new Grid2D(List.of(1,2,3), 2).add(4).add(5)
$.. ==> {1,2;3,4;5}
```

Level 3

Include the `get` and `set` methods in `Grid2D` to get an element, and set the element of the two-dimensional grid respectively.

- the method `get(int r, int c)` returns the element from row `r` and column `c`.
- the method `set(int r, int c, int elem)` sets the element at row `r` and column `c`, and returns the new `Grid2D` object.

You may assume that row `r` and column `c` are valid positions on the two-dimensional grid with some element value.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> new Grid2D(List.of(1,2,3,4,5), 2).get(0,0)
$.. ==> 1

jshell> new Grid2D(List.of(1,2,3,4,5), 2).get(1,1)
```

```
$.. ==> 4

jshell> new Grid2D(List.of(1,2,3,4,5), 3).get(1,1)
$.. ==> 5

jshell> new Grid2D(List.of(1,2,3,4,5), 2).set(1,1,9)
$.. ==> {1,2;3,9;5}

jshell> new Grid2D(List.of(1,2,3,4,5), 3).set(1,1,9)
$.. ==> {1,2,3;4,9}
```

Level 4

We are now ready to define the `Puzzle` class with the following specifications:

- define a constructor `Puzzle(int n)` that takes in a integer value `n` and creates a `(n`^{2}` - 1)` puzzle in a solution configuration.
- define a `toString()` method that outputs the puzzle.

Each tile should be output with leading spaces using the format `String.format("%4d", ...)`. In addition, output a dot `.` to represent the empty tile.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> new Puzzle(4)
$.. ==>
   1   2   3   4
   5   6   7   8
   9  10  11  12
  13  14  15   .

jshell> new Puzzle(5)
$.. ==>
   1   2   3   4   5
   6   7   8   9  10
  11  12  13  14  15
  16  17  18  19  20
  21  22  23  24   .
```

You may assume that the puzzle will be at least of dimensions `2`-by-`2`.

Level 5

Sliding the tiles of the puzzle is implemented by specifying which tile is to be moved. Define a method `Puzzle move(int num)` that takes as argument the number of the tile to be moved.

A tile with number `num` can be moved if there is an empty space above/below it, or to the left/right of it. If a move is possible, make the move and return the resulting `Puzzle` object. Otherwise, the existing puzzle is returned.

In addition, include the `isSolved()` method that returns `true` if the puzzle is solved, or `false` otherwise.

You may assume that `num` is always in the range of 1 to `(n`^{2}` - 1)` inclusive.

Note that the following sample run shows the puzzles moves for a `4`-by-`4` puzzle only.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> Puzzle puzzle = new Puzzle(4)
puzzle ==>
   1   2   3   4
   5   6   7   8
   9  10  11  12
  13  14  15   .

jshell> puzzle.move(10)
$.. ==>
   1   2   3   4
   5   6   7   8
   9  10  11  12
  13  14  15   .

jshell> puzzle.move(12)
```

```
$.. ==>
   1   2   3   4
   5   6   7   8
   9  10  11   .
  13  14  15  12

jshell> puzzle.move(12).move(11).move(15).move(12)
$.. ==>
   1   2   3   4
   5   6   7   8
   9  10  15  11
  13  14  12   .

jshell> puzzle.move(12).move(11).move(15).move(12).isSolved()
$.. ==> false

jshell> puzzle.isSolved()
$.. ==> true
```