Homework 1 :: Column Puzzle

15-121 Fall 2015
Due: September 08, 2015 (11:50pm)
JAR, Basecode, Hand-In

Overview

In this homework, you will be implementing a column puzzle. A column puzzle is a toy which has 6 columns of colored plastic tiles. One tile in one column is missing so other tiles can be moved horizontally or vertically into the empty space. The column puzzle is solved when each column consists of the all the tiles of the same color.

As with all assignments, you will be graded in part on your coding style. Your code should be easy to read, organized, and well-documented. Be consistent in your use of indentation and braces. See the style guide for more details. For this homework (and homework 2), style penalties will only be half as severe as usual to give you time to learn the syntax of the language and to become familiar with the style we expect.

Background :: Code

For this homework, you will be working with a 2-dimensional array (a list, if you are coming from Python). You can think of this 2D array as a grid, where each point is a coordinate starting from [0,0] in the upper left corner. Methods have already been provided in the base code that use the 2D array to "paint" the column puzzle for you. You only need to modify the 2D array. Changing the color of any position in the `colors` array will cause the column puzzle to change when repainted. Clicking on a colored square adjacent to the blank square will swap the colored square with the empty square (colored in black).

Run the example JAR file to understand how the game is played. Although the example JAR file only shows a puzzle with 3 rows, your code must work for any specified number of rows. There will always be 6 columns, denoted by the global variable `NUM_OF_COLUMNS`. You should use the global variable `NUM_OF_COLUMNS` in your code, not the literal number 6.

Exercise :: Identify

Download the base code and fill in the required fields like so:

```
/**
 * @author [First Name] [Last Name] <[Andrew ID]>
 * @section [Section Letter]
 */

/**
 * @author Jess Virdo <jvirdo>
 * @section B
 */
```

Inspect each of the methods that have been already implemented for you. You are not responsible for understanding each line of code, but you should understand the overall algorithm and concept. We recommend that you read through this entire spec and follow along in the base code before you start writing. This will help you have a better understanding of how the code is structured.

Exercise :: setup

The `setup` method initializes the Column Puzzle. You must use the following algorithm in your implementation:

1. Initialize the `Colors` array `grid` with the specified number of rows and 6 columns.
2. Initialize each column with the `Color` corresponding to the same index in `colorsList`.
3. Set the bottom-right cell to black, since it is the cell we will use to swap the two adjacent squares.

Exercise :: swap

Implement the `swap` method that swaps two adjacent squares in the `colors` array.

Exercise :: shuffle

Complete the shuffle method. You must use the following algorithm:

1. Create a new `Random` object. You may need to consult the Java API for the Random class.
2. Pick a random color location in the `colors` array (by choosing a random row and a random column).
3. Swap this color with another color at a different location. You must use the `swap` method you wrote previously.
4. Repeat this process until you have performed 100 color swaps. (You do not have to create a new Random object for every swap -- you just need to swap 100 times.)

Exercise :: setTitle

Complete the `setTitle` method that updates the title on the `Frame` window. Consult the GridGUI file.

You must complete this method in one line of code.

Exercise :: isSolved

This method returns true if and only if the puzzle is solved, and returns false otherwise. The puzzle is solved when each column consists of exactly one color (and potentially the blank tile).

Exercise :: adjacentBlackSquare

Complete the `adjacentBlackSquare` method. If the black square is adjacent to the given square, this method returns the (x,y) coordinate of the black square (in an `int[]` array of size 2). If the black square is not adjacent to the given square, this method must return `[-1,-1]`.

Exercise :: mousePressed

Implement the `mousePressed` method using the following algorithm:

1. Find an `adjacentBlackSquare` to the clicked square.
2. If there is no adjacent black square, update the title to say "Illegal move" and halt execution.
3. Otherwise, `swap` the black square with the clicked square.
4. Update the total number of moves.
5. Update the title. If this move solved the puzzle, update the title to say "You won!". Otherwise, update the title to say "X moves", where X is the number of moves so far.

Exercise :: Test

For this homework, we have given you a tester file. This tester will give you an idea of what is expected from you and how we may grade your work. Some of the methods in this tester are unimplemented. You are strongly encouraged to fill them in, though you're not graded on your tester and you should not hand it in. In the future, you will have to write your own tests from scratch.

Submitting your Work

When you have completed the assignment and tested your code thoroughly, create a `.zip` file on your work. Only include the following file(s):

1. `ColumnPuzzle.java`

Do not include any `.jar` or `.class` files when you submit, and zip only the files listed above. Do not zip not an entire folder containing the file(s).

Once you've zipped your files, visit the autolab site -- there is a link on the top of this page -- and upload your zip file.

Keep a local copy for your records.