CS 1331 Homework 8

Due Thursday, October 25th, 2012 8:00pm

Introduction

This assignment will give you practice with 2D arrays, layout managers, actionlisteners, and class design. You will get this experience through creating an "Eight Puzzle" game (a smaller version of the more commonly known 15-puzzle game).

An 8-puzzle is a sliding puzzle that consists of a frame of numbered square tiles (or pictures) in a jumbled order, with one empty space. The puzzle we will be creating has a size of 3x3 tiles and the object of the puzzle is to place the tiles in order by making tiles adjacent to the empty space "slide" over into that empty space.

Please note, we will post an 8-puzzle solver later to help you in testing your game by stepping through the solution.

Scrambled state

Solved/Start State

Do not forget about javadocing and commenting detailed in previous homeworks.

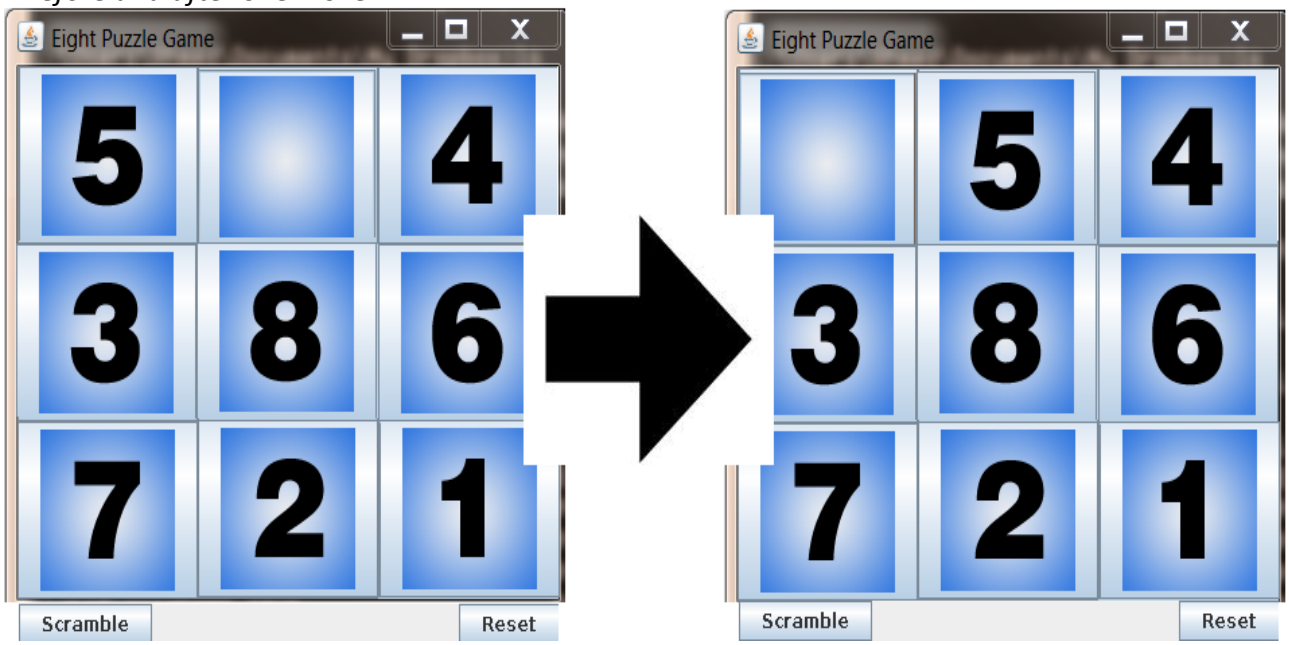
8.0 Eight-Puzzle

The game follows some simple rules:

- The game is made up of a square board with a 3 x 3 tile arrangement.
- There will be 8 defined pieces and one empty tile.
- The game starts when the tiles laid out in order (1-8, blank spot bottom right) (see

- The game begins in this start state, you should have both a reset button (to return it to the start start) and a scramble button (which randomly scrambles the tiles). **Please note the the images provided are missing these buttons**.
- A numbered tile can only move when it is adjacent (to the left, right, above, or below) the empty tile.
- A tile cannot move diagonally into the empty slot.
- The game is won when the tiles end up back in their correct order as shown here:

Before and after one move:



You will be responsible for making a Piece, EightPuzzle, EightPuzzlePanel, and EightPuzzleGame class. We are trying to introduce some good object-oriented design by having you create separate classes for different elements of the game. The Piece class will provide a template for how a tile in the 8-puzzle game will behave. The actual game logic will be implemented in the EightPuzzle class. The EightPuzzlePanel class will use the state of the EightPuzzle to draw the parts of the game on a JPanel, and respond to action events from the user to send as input into the game logic. Finally, the EightPuzzleGame class will create an instance of EightPuzzle and provide a JFrame and a main method in order to display the GUI of this 8-puzzle game.

Important: In the instructions, you will notice some of these implementations are the suggested approach and alternatives have been listed. The key aspect is we are grading on is functionality - does it work in the way we asked it to? - and good design.

This means, if you want your piece to store an int expected x and int expected y, a Point expectedPoint, or even a 2 element array coordinate pair - that is fine. As long as your code works **and** makes sense. Hence, having row 1, col 1 be the bottom left is fine (although changing the indexing and order of the dimensions make manipulating the array difficult).

However, note the second stipulation **good design**. First and foremost, this means good object oriented design. Hence, having a hard-coded catch all ActionListener is not equivalent to having individual ActionListeners, which itself is not equivalent to having individual ActionListeners objects instantiated from a general, parameterized class. This is just one example of a design flaw, but there are many other areas you could go wrong, be sure to utilize the resources you have available. Another aspect of proper class design is that your piece class should be able to be used for purposes other than a traditional 3x3 sized puzzle

Homeworks that do not run will not receive credit. A homework that throws a null pointer error the second it starts, thereby crashing everytime, is the functional equivalent of a homework that does not compile and will therefore also receive a grade of 0.

8.1 Piece.java

This first class will represent a single Piece that is going to be used on the 8-Puzzle game board.

1. Instance variables
   - **final int VALUE** will be the value of the piece - must be a constant
   - **ImageIcon image** to store an ImageIcon object that will hold a specific image for the Piece
     - This just has to be some sort of visual representation of the piece, you can choose to not use an image but we have provided some for you.
   - **Suggested:**
     - **final int EXPECTED_COL** will give the final column position of the instance of Piece
     - **final int EXPECTED_ROW** will give the final row position of the instance of Piece
       - Using a Point (java.awt.Point) is acceptable here. Or X and Y, etc....
     - Essentially, you really don't technically need these variables stored in the Piece, as long as you have some other way to properly evaluate the win condition (when all the pieces in the game are in their correct position)
   - Remember to enforce proper encapsulation throughout the assignment.
2. Create a constructor that creates the object and assigns its proper fields in accordance with your game logic.
   - **public Piece(ImageIcon image, int col, int row, int value)**
     - Do not assume anywhere that the board size is 3x3. This is a generic piece object, and it should work just as easily for a 15 Puzzle (a 4x4) or any other Puzzle Game (ie 11 puzzle - 4x3)
3. Create getters for all of the instance variables
4. Remember, the EXPECTED_COL and EXPECTED_ROW represent the col and row position of the piece where it should be for a completed solution. For example, the 1 piece should be at row 0 and col 0 since it is in the top left corner. If you do choose to stick with this notation, this will match up perfectly a 2 dimensional array (row index 0, col index 0 = 1st piece).

8.2 EightPuzzle.java

The EightPuzzle class will create a game board using a 2D Array to hold all of the game's Pieces. Here is where you will be creating board in order to start the game and adding the functionality of being able to make a move, randomly scramble the board, and check for a win condition. This class abstracts the actual logic of the 8 puzzle game away from the GUI, and consequently the user. Therefore, the only way the internal state of the game should be able to change is through your public facing methods.

If you follow proper encapsulation, your game should never crash due to bad user input.

1. Create at least these instance variables, two of which are constants:
   - **final int BOARD_SIZE** will give the size of the board on each side, which is 3
   - **final int SHUFFLE_NUM** to hold the number of times to swap Pieces during shuffling
     - Remember to use these constants in your code - do not just hardcode in numbers after you have already created these variables for those values.
   - **Piece[][] board** is a 2D array that will hold all of the game's Pieces
   - A **Random** object in order to randomly choose Pieces of the board to swap.
   - **Suggested** to help with logic:
     - **int currentCol** will hold the current column of the blank Piece
     - **int currentRow** will hold the current row of the blank Piece
2. You should have a **constructor** that will instantiate the board array with a 3x3 array and populate it with he game Pieces 1-8 starting in the top left and going across, with the empty square in the bottom right corner.
   - *How you to choose to represent the blank piece is up to you, but will have significant impact on you end up displaying and moving the pieces.*
   - Make sure you use **relative file paths** or else your code will not compile on someone elses computer.
   - Initializes any other variables pertinent to your logic.
3. A shuffle method, **public void scramble()**, that randomly, validly rearranges the pieces in the puzzle in order to create a **more** scrambled puzzle to solve.
   - You should be able to further scramble an already scrambled board.
   - You must make sure that your scrambling algorithm follows good statistical random distributions attributes.
   - You must also use a high enough **SCRAMBLE_NUM** to ensure you have a properly shuffled board.
   - There are a number of ways to perform scramble that will depend on how you implement your move method, but I will suggest a simple one later.
4. a move method that returns a boolean, **public boolean move(?):**
   - This method must take in a proposed move that the player is attempting to make. If it is a valid move, the game performs that move and returns true. If it is an invalid move, the method returns false.
   - The only valid moves are those involving moving a puzzle piece and the empty piece. Whether you wish to conceptualize it as a swap of 2 pieces or the physical movement of one piece into an empty position is up to you.

- You may define the parameters to this method however you wish. It can take in an x,y coordinate pair of the destination square, the location of the square to be moved, or simply a single value corresponding to one of the 4 possible movements - UP, DOWN, LEFT, RIGHT.
- Any invalid move, ie trying to move a piece outside the bounds of the puzzle board or moving the 3 piece into a slot already occupied by another piece, should return **false but not** throw an error.

5. A **public boolean isSolved()** method which returns true if the puzzle arrangement is solved (in ascending order). Essentially this method evaluates whether the win condition has been met
6. A **public void reset()** method that returns the board to its initial state where all the pieces are in their correct location.
7. A public **getGameView** method that will return a visual representation of the current state of the game. This method will be called by the GUI so that it can update its representation of the current state of the game. Hence, the return type of this method will changed based off of your implementation.
   - The must important aspect of these getter methods is that they should not return anything which would could be used to alter the state of the game. Hence, when dealing with arrays, it is always recommended you return a deep-copy of the array rather than the actually array itself.
   - If you followed the ImageIcon stored in the Piece object design, your method could possibly be:
     i. **public ImageIcon[][] getGameView** - returns a 2 Dimensional Array of ImageIcons representing the physical layout of the puzzle
8. Any additional methods you may need to use, both public and private. Just make sure any addtional public methods do not violate the encapsulation (and hence stability) of the game.
   - An example of a useful public method you might want to have, might be something like getEmptySquareLocation

8.3 EightPuzzlePanel.java

The EightPuzzlePanel class will extend JPanel in order to display and run the game. It should hold an instance of your EightPuzzle class and display a 3x3 array of buttons or panels in a grid which will correspond to the pieces of the puzzle. It should also in have a reset and scramble button.

When the game starts out, it will be displayed in its original solved position. Pressing the scramble button at any point will scramble the puzzle further. Pressing reset should return it to its start/solved state. The user should be able to click on any square in the puzzle. If that square corresponds to a valid move, the move is performed and then the GUI is updated to represent the change. After each move, the game should be checked to see if the puzzle is solved. If it has been solved, a dialog should be displayed indicating so and prompting the user whether they wish to continue or quit.

- The EightPuzzle game itself must be stored as an instance variable, and taken in by the constructor.
- You must use arrays of component, efficiently arranged in grid, displayed on the panel which will display the 'pieces' of the puzzle.

- There must be both a scramble and a reset button somewhere on the panel. I suggest using nested panels to organize the layout.

1. Include a constructor **EightPuzzlePanel(EightPuzzle puzzle)** which
    - saves the EightPuzzle instance
    - creates all of the buttons, initializes them, creates action listeners for them, and adds them all to the JPanel.
    - Alternatively, you can use a different JComponent and associated listener.
    - Creates, sets up, and adds a Reset Button beneath the puzzle, which will reset the game when clicked
    - Creates, sets up, and add a Scramble Button beneath the puzzle, which will further scramble the puzzle once clicked.
    - Performs any other necessary steps so that the game is playable upon creation, such as possibly calling the following updateGUI method.

2. A method **updateGUI()** which changes the image displayed by each of the buttons (or whatever is applicable to your solution) to match the internal state of the EightPuzzle game. This method must utilize the **getGameView** method from the EightPuzzle class and be called wherever and whenever necessary.
    - Please note there is an updateUI method in JPanel, which is meant to perform this function. If you end up calling your method updateUI as well, please make sure you place a call to super.updateUI() in the first line of code.

3. A private inner class which implements ActionListener (if you used JButtons). A full credit action listener design is outlined:
    - This class should have two instance variables **int col, row** to keep track of its location on the board. (or x,y)
    - In turn, the constructor should be parameterized, ie **ButtonListener(int col, int row),** to assign the instance variables.
    - ActionListener classes which do not use a parameter but rather have a hard-coded response involving some kind of conditional flow control will not be awarded credit. (Partial credit for writing individual specific classes)

    Logic
    - When a button is clicked, the GUI should determine the appropriate command that needs to be called. This means it should EightPuzzle's move method and calculate any of the necessary parameters.
    - If the move method returned true, you should update the GUI.
    - The listener should check to see if the puzzle is solved. Upon winning, you should display a confirmation dialog which prompts the user if they want to play again. If they choose to quit, you should terminate the program. (Or optionally including a 3rd choice to scramble the puzzle and continue).

8.4 EightPuzzleGame.java

You need a game 'driver' to actually run the code. The EightPuzzleGame class will create the JFrame and perform all the operations required for creating and running a swing application, including creating the EightPuzzle object and then passing it in as the parameter when it creates a EightPuzzlePanel. This class should only have one static main method.

Some Hints:

- **Scrambling**
  Many of the best algorithms are those modeled after our own natural thought process. When trying to decide how to scramble a puzzle, how would you do it? Randomly pick squares and see if they correspond to an valid new arrangement? Would you waste your time enumerating all the possibilities if you were doing this in real life with a physical puzzle?

  Essentially, you only ever have 4 possible actions in this game. To move any square up, down, left, or right. The movement of each square is going to be restricted by the locations of the the other squares and the boundaries of the board. When you look at a puzzle you are holding in your hand, you instinctively understand those squares aren't capable of being moved. On a computer, there is no way to know that but to check each square and the conditions around it.

  However, what about the 'empty' square? If every valid move involves the empty square location, isn't possible to just imagine that as the piece you are attempting to move? Then at most, there are only ever 4 possible moves to choose from and at most 4 legality checks which are now reduced to bounds checking.

  Think about how you might even modify your move method input parameter to simplify the representation of the 4 possible moves. You may want to consider looking into *Enums* or uses of public constants.

- Action listeners can be parameterized – they are objects like everything else, so you can create a constructor for your private inner class that can take in a row and column (this will make the array of components and their corresponding action listeners very easy to navigate).
- You are not restricted to the ActionListener class to respond to user movement's, nor are you restricted to using JButtons. If you wish to use a mouse listener with an array of **individual** panels, that works as well.
- **LayoutManagers:** You can nest panels inside eachother, with subpanels having their own layout managers. Considering reading up on Grid Layout, FlowLayout, and BorderLayout.
- For those of you who do use JButton, consider looking up *inherited* methods such as setIcon in the API http://docs.oracle.com/javase/6/docs/api/javax/swing/JButton.html
- **JOptionPane**
  Once a win condition has been determined, in order to prompt the user to continue playing, reset, or quit, I highly suggest reading the API page for JOptionPane. Here is a great tutorial which shows some examples: http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html

- Nested for-loops are your best friends when it comes to 2D arrays

2D Arrays and Nested For-Loops

Remember, 2D arrays are really just an array of arrays. If you want to think about it in diagram format it would look something like this for a 4x4 array.

```
[ ] -> [ ] [ ] [ ] [ ]
[ ] -> [ ] [ ] [ ] [ ]
[ ] -> [ ] [ ] [ ] [ ]
[ ] -> [ ] [ ] [ ] [ ]
```

The red spaces are the first array. Each slot in the first array holds another array (the black ones). If we want to iterate through this, the best way is to use nested for loops, like this**.**

The most consistent problem with students dealing with a 2D Array in a gui is the difference between row/col and x/y notation. Mainly, the row is actually a vertical dimension, which means it is a y coordinate, not x. Always be aware of this difference in writing your logic.

Take note of how [row][col] is different from [x][y] of you were using that visualization. If you do choose to use x,y coordinates in that fashion, then you might want to consider saying [y][x] instead

Images: We have provided you with some default images or you may use your own. Whatever you choose, make sure you submit the images with your submission and **use relative file paths**.

We will provide you with an autosolver later for generic eight puzzle games to help speed up the testing of your game. For right now, focus on the logic of the game. You can test the win condition by starting with a reset game and manually scramble it by choosing which tiles to move, and then reverse it.

Turn-in Procedure

Turn in the following files to T-Square. When you are ready, make sure that you have actually **submitted** your files, and not just saved them as a draft.

- *Piece.java*
- *EightPuzzle.java*
- *EightPuzzlePanel.java*
- *EightPuzzleGame.java*
- Any images you used, even if you chose the ones provided.

Note** Always submit .java files - never submit your .class files. And be 100% certain that the files you turn in compile and run - submissions that do not will receive an automatic 0. Also, make sure that your files are in on time; the real deadline is 8 pm. While you have until 2 am to get it turned in, we will not accept homework past 2 am for any reason. Don't wait until the last minute!

Verify the Success of Your HW Turn-in

Practice "safe submission"! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.
2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.
3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.
4. Recompile and test those exact files.
5. This helps guard against a few things.
    a. It helps insure that you turn in the correct files.
    b. It helps you realize if you omit a file or files. (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)
    c. Helps find last minute causes of files not compiling and/or running.