Project 1: Dots!

Note on compiling Board.java

You will not be able to compile Board.java until you make your own CantRemoveException class. (See `removeSelectedDots`)

Note on compiling GUI.java

If GUI.java fails to compile because of some complaint about the `validate` method call, replace all occurences of `validate` with `revalidate`. Do not do this if GUI.java compiles fine as is.

Background Information

In this project you will be implementing a simplified version of the popular smartphone app Dots (the app is free, go try it out yourself for a feel of what it's like) as well as creating a simple AI searcher. To earn full credit, you must correctly implement all provided methods (`// your code here` areas) unless they are marked "optional" in the skeleton file. Feel free to add your own custom methods and instance variables. Do not modify any `public static final ints`.

The goal of this game is to score as many points as you can given a restriction on the number of moves you can make. You must connect at least 2 adjacent dots of the same color to make them disappear. Dots must be connected one at a time, and each dot selected after the first of each move must be adjacent (up, down, left, right) to the previously selected one. If the dots are connected such that they form a *closed shape* (Fig. 1), then all dots of the same color as the connected dots will be removed. You earn one point in Dots for each dot you remove. After each turn, the remaining dots will drop if they are above an empty space left behind by the disappearing dots. Then, randomized dots will drop from above to fill up the remaining space.

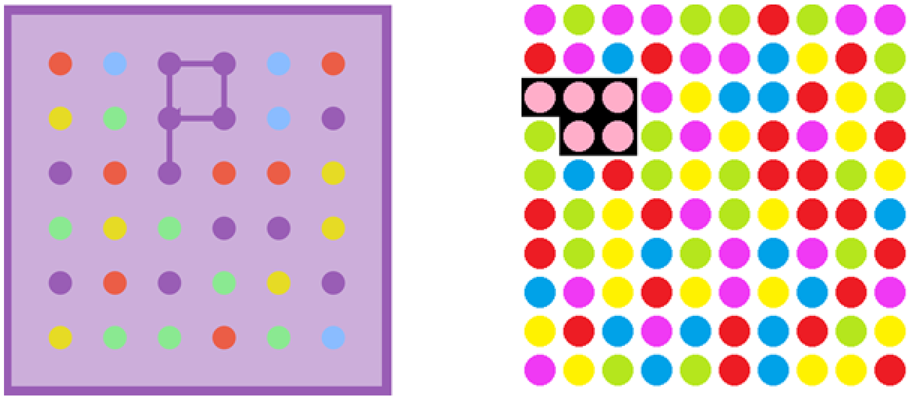
Fig 1

On the left is what a closed shape looks like on the actual app. The region on the right is what your project's

closed shape will look like.
Note: A closed shape must consist of 4 or more dots such that the path reconnects to itself.

Helper Files

- We have included a GUI file (`GUI.java`), which works with the skeleton files (`Board.java` and `Dot.java`) to create a functioning game with a graphical interface. Do not modify this file.
  Unless the staff has instructed you to do so, **DO NOT MODIFY THIS FILE.** You may modify the file to change validate to revalidate as mentioned at the top only if necessary.
- To accompany the GUI, we included a folder of images. Do not rename this folder or modify its contents. If you are using Eclipse, keep this folder in your Project Folder (i.e. the same folder as src and bin, if applicable). Otherwise if you are using the command line to compile and run your project, keep the images folder in the same folder as your .java and .class folder.
- To test your code with the GUI, run the main method in Boards.java. To test without the GUI, create your own JUnit tests.

Part 1: Board Setup and `select` methods

Our board will be represented by an $n \times n$ `Dot[][]` named myBoard, where n is the number of dots along a side of the board. Colors will be represented by `public static final ints` (positive integers). An example of the board's coordinate system can be seen in Fig 2. `myBoard[1][2]` will point to the blue dot with a red X on it.

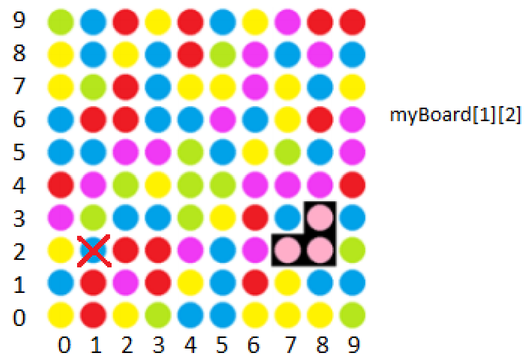

Fig 2

`Dot` will be a small class you must implement that represents the dots on the board.

`Board` Constructor

Takes in an int size and if the size is valid, it fills our board up with random colored dots (See Dot constructor) for each entry. Make sure the argument *size* is within the minimum and maximum size constraints. If not, have the program send an error message (System.err.println("Your message here")) and exit (System.exit.(1)).
There is also another constructor which takes in an `int[][]` of colors such that the colors match the coordinates in the diagram above. You may assume that the input is valid (between MINSIZE and MAXSIZE etc.) since this is for your own testing.

`movesAllowed`

`movesAllowed` is a `static int` that describes how many moves are allowed in a single game. There are setter and getter methods for this variable to be implemented.

`getMovesLeft`

Your "moves left" should initially be equal to `movesAllowed`. A move is defined to be the number of times you successfully remove dots. That is, each time you click the "Remove" button, if 2 or more dots are successfully removed, the result from getMovesLeft() decreases by 1. When you have 0 moves left, the game is over.

`canMakeMove`

Returns if it is possible to make a move with the board's current arrangement. If the user ends up with a very unlucky setup either at the beginning or after a move is made, they are unable to make any moves and the game ends (handled by GUI). You can only make a move if there is at least one pair of adjacent dots of the same color.

`isGameOver`

Returns if the game is in a state of game over. The game is over if there are no possible moves left or if the player has used up the allotted moves.

`canSelect`

Called when the user attempts to click on a dot. If true, the dot will be highlighted and thus "selected". A move is only legal if it is both adjacent to and the same color as the previously selected dot. If the game is over, then no dots can be selected. If no dots are selected and the game is not over, then any dot selection is legal.

`canDeselect`

Called when a user attempts to click a selected dot. You can only deselect the most recently selected dot.

`selectDot` and `deselectDot`

Called when the user clicks on a dot. If the dot at (X, Y) has not been selected yet, `selectDot` is called. Otherwise, `deselectDot` is called. These methods are only called if `canSelect` or `canDeselect` return true respectively. If you click on an unselected dot for which `canSelect` returns false, nothing happens. Similarly if you click on a selected dot for which canDeslect is false, nothing happens.
Also, the GUI will handle the fact that `selectDot` is only called if `canSelect` is true for that dot. The same goes for `deselectDot` and `canDeselect`.

`Dot` Class

This class represents the dots that appear on a board. The Constructor should create a dot with a random color out of the provided `public static final int` variables. When keeping track of a dots color, remember that the color is actually an `int` type.
Note: There is a variable defined as NUM_COLORS which should be used for generating random colors (ints). You random number generator should return an integer from 1 to NUM_COLORS inclusive, (not 1 - 5). This is good coding practice for scenarios like when you end up changing one variable (such as NUM_COLORS) and don't want to make yourself of others fish through their code replacing every 5 with a new number.

Part 2: What happens when you click Remove?

When you click the Remove button a series of events will occur. Assuming the removal of dots should happen (see RemoveSelectedDots), the selected dots will "disappear", gravity will be applied to the board simulation, and then random colored dots will fall from above until the board is filled up again.

Here is a diagram of the process:

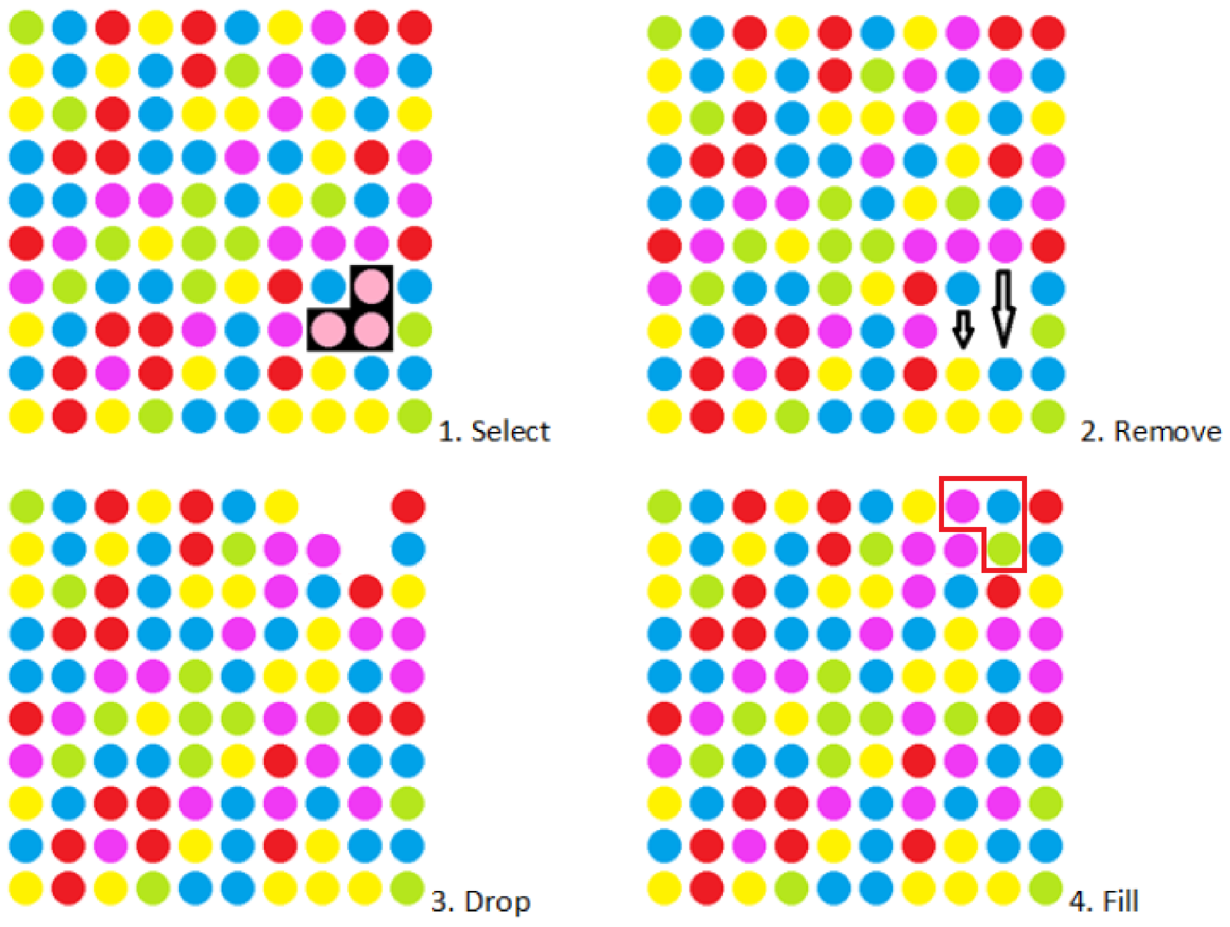

Fig 3

`removeSelectedDots`

Called when the user clicks the Remove button. If no dots are supposed to be removed, (i.e. less than 2 dots are selected) a `CantRemoveException` is thrown. You must create a `CantRemoveException` class in order for your code to compile. To do so, simply make a `public static class` within your Board.java file. If the remove action is successful, this method will change state of all Dots of the same color as the selected dots to a "removed" state.

`isClosedShape`

Return whether or not the selected dots form a closed shape. See Fig1 above for an example. In Fig 1, 5 dots are selected and they form a closed shape. If there is a `closedShape`, `removeSelectedDots` will remove all dots of the same color as the selected dots instead of only the selected dots. NOTE: A shape is only considered closed if the most recently selected dot is adjacent to at least 2 other selected dots, indicating that it was linked from one dot and is able to link to another dot that was already selected.

Fig 4 shows two shapes that look the same, but one is not a closed shape due to the order of selection. If the last selected dot is not able to link to a previously selected dot, no closed shape is formed.

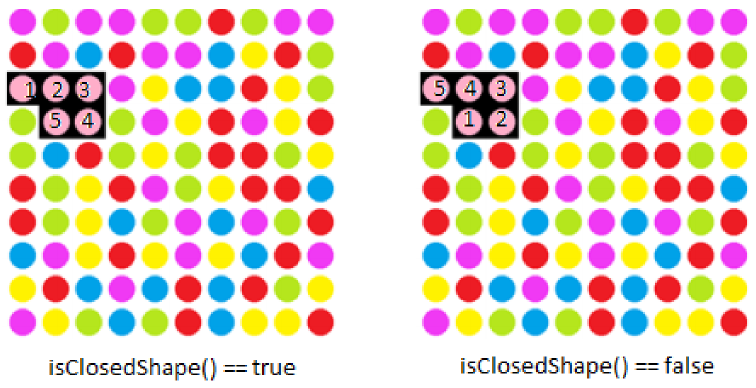

Fig 4

`removeSameColor` (optional)

This is an optional helper method. If the selected dots form a closed shape, this can be called by `removeSelectedDots` to remove all dots of the same color as the selected dots.

`dropRemainingDots`

Once the dots are removed. Rearrange the board to simulate dropping of all of the dots that are not in a "removed" state. For instance, if there is one "removed" dot in a column, all dots vertically above this dot will shift down by one. You can handle the topmost dot however you like as long as fillRemovedDots replaces it with a randomly generated dot.
Note: `fillRemovedDots` will make one replacement for each dot that should be removed. These replacements will be random colored dots.

`fillRemovedDots`

After removing all dots that were meant to be removed and simulating the dropping of dots, this method will replace the "removed" dots on top with new random dots to simulate refilling the board. (Refer to diagram).

Part 3: AI

The AI portion of this project is a square detector. If a square exists, it will find the optimal square to choose out of all other squares.

Implement the method `findBestSquare` to search the board for a sequence of 4 points which form a square of length 2 such that out of all possible squares, selecting this one yields the most points. Return an `ArrayList` of `Point`s whose sequence is a legal order that satisfies conditions above. If there are no squares present, return null.

Note that the actual Dots game removes all dots of a color with the selection of closed shapes other than just squares. In this project, we only require that you detect squares of side length 2.

AI: Challenges

These might be fun for those who have had prior experience working with graph algorithms. Neither of them will be graded.

- Implement a `findBestBigSquare` method that finds the best square of length at least 2 that removes the most number of dots currently on the board. All dots on all 4 sides of the square must have the same color.
- If you're really ambitious, implement a `findBestClosedShape` method that finds the best closed shape of the same color that removes the most number of dots currently on the board.

Helpful tips

When working on the project consider the following (These are just tips. You do not have to follow them if you don't want to):

- Represent selected dots with as Point objects.
- Have a method or instance variable keep track of the most recently selected point.
- Have your own custom instance variables and methods to make your other methods easier to implement.
- Use the optional methods provided in the skeleton.
- To generate random numbers, look into (google) java.Random and Math.Random.
- For `dropRemainingDots`, you can make it so that all "removed" dots are placed on top, which may make your `fillRemovedDots` method easier to implement.
- In your Dots implementation, you may choose to add instance variables to represent your dot's coordinates, but be warned, these variables must be consistent with their position in the `Dot[][]` myBoard, or errors may occur.
- Start early! Give yourself extra time in case you come across bugs. We have a checkoff on Monday 7/7 to encourage you to start early.

Checkpoints

The following features can be tested using the GUI given that the required methods listed below them are properly implemented:

Each feature is dependent on all methods listed in previous features.

New Game

- Board.getMovesLeft
- Board.getScore
- Board.getBoard
- Board.isGameOver
- Board(int n) Constructor
- Dot.getColor
- any methods that these methods are dependent on

Select/Deselect Dot

- Board.canDeselect
- Board.deselectDot
- Board.canMakeMove
- Board.canSelect
- Board.selectDot
- any methods that these methods are dependent on

Remove

- Board.removeSelectedDots
- Board.dropRemainingDots
- Board.fillRemovedDots
- any methods that these methods are dependent on

Best Square

- Board.findBestSquare
- Board.numberSelected
- any methods that these methods are dependent on

Project Checkoff (due July 8)

There will be a checkoff in lab on Tuesday July 8.

Make sure you correctly implement the required methods so that you can successfully use the first two features: New Game and Select/Deselect Dot.
Tip: When your files pass the checkoff, save a copy of it in case it breaks when you try implementing other features.

Submission Information

Create a `readme.pdf` or `readme.txt` file that includes:

- the names and logins of all group members
- a description of each of your test's purpose
- a short summary of each member's contribution to the project

Submit your solution to this project by Saturday July 12, 10pm PDT with the command `submit proj1`.
Your submission should include the following files:

- `Dot.java`
- `DotTest.java`
- `Board.java`
- `BoardTest.java`
- `CantRemoveException.java`
- `readme.txt` / `readme.pdf`

in addition to all other Java files (including test files) you wrote for this project. Only one group member needs to submit per group.

Grading

This project is worth 15 course points. The grade breakdown is as follows:

- Code correctness: 8
- Comprehensive testing: 5
- Proper Java conventions and code style: 1
- `readme.txt`: 1