Com S 227
Fall 2017
Assignment 3
300 points

Due Date: Thursday, November 2, 11:59 pm (midnight)
"Late" deadline (25% penalty): Friday, November 3, 11:59 pm

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html **, for details.**

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy acknowledgement* on the Homework page on Blackboard. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

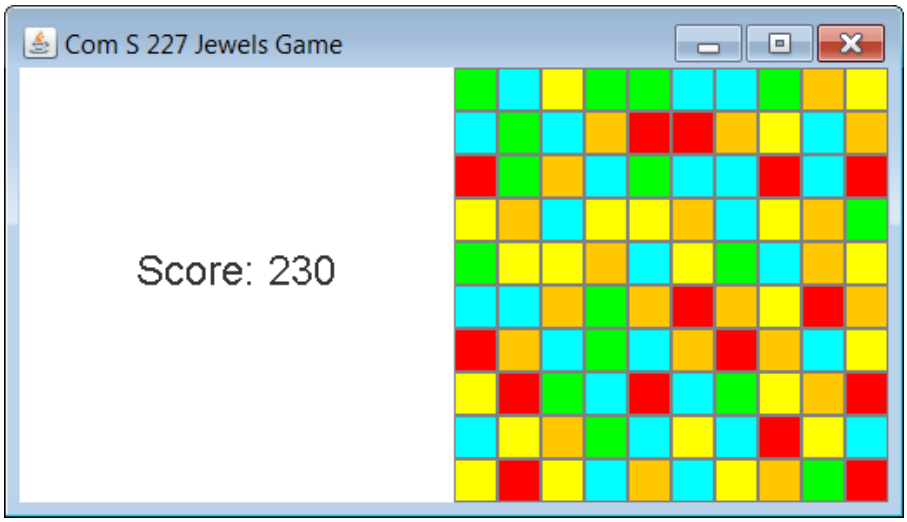
*Our second exam is Thursday, November 9. Note that it will **not** be possible to get all the homework submissions graded before the exam.*

Please start the assignment as soon as possible and get your questions answered right away!

Introduction

You will be writing the backend logic for a simple version of a video game sometimes known as "Bejeweled". You'll get lots of practice working with arrays, 2D arrays, and ArrayLists. Once you have your code working, you'll be able to integrate it with a GUI that we have provided to create a complete application.

Your task will specifically be to implement the three classes `Game`, `JewelFactory`, and `Util`. These classes go in package `hw3`, and you should not modify anything outside this package (except possibly to edit `ui.GameMain`). There is a skeleton of the `Util` class given.



The game is played on a grid of colored squares or icons that we will call "jewels". The objective is to amass points by making *runs*, where a run is defined as 3 or more adjacent, matching icons in a row or column. You can click and drag from an icon to an adjacent icon (horizontal or vertical) to try to exchange them. This is called *selecting* the pair. If exchanging the two icons creates one or more runs, the selection *succeeds* and the icons are exchanged; otherwise nothing happens. When a selection succeeds, all icons that are part of a run disappear. Then each column *shifts down*, that is, any icons that are above the empty cells move down (as if by gravity) into the empty spaces, and new ones *fill* the empty cells (that are left at of the top of the column). Note that because the new cells are randomly generated, runs may be formed as a result of the new cells being added to the grid. Again, all cells that are part of a run disappear, the columns shift down, empty cells are filled, and this process iterates until the grid contains no runs. Points are added to the score for each run that is formed.

You can take a look at http://en.wikipedia.org/wiki/Bejeweled for an overview of the original game (there are also various sites where you can try it online). Our version will be simplified, however.

Detailed specification

Carefully read the javadoc for details about the precise behavior of methods. You can find all the documentation online at http://web.cs.iastate.edu/~cs227/homework/hw3/doc/. The sample code includes a partial implementation of the class **`Util`** located in the **`hw3`** package. A few methods are already implemented to help get you started. You'll also need to be familiar with the two (very simple) classes in the **`api`** package: a **`Jewel`** encapsulates a single integer, representing a color or type of icon, and a **`GridPosition`** encapsulates a **`Jewel`** along with a **`row`**

and **column**, and is used to represent the contents of one position in the game grid. There is also a package **ui** containing code for a sample GUI, discussed later.

There are also some usage examples later in this document in the "Getting started" section.

Overview of the Util class

As its name implies, **Util** is a "utility" class, meaning that it is just a collection of static methods performing various useful functions. The key algorithms for the game are isolated in this class in order to make them easier to develop and test independently as one-dimensional array operations independent of the game itself. These are a) the algorithm for finding runs and b) the algorithm for shifting non-null cells to the end of an array. You are required to implement these two algorithms in the **Util** class and not duplicate the code in the **Game** class. (The **Game** class includes methods for copying a row or column into a one-dimensional array and back again, so it will be easy to use the **Util** methods to operate on a row or column of the game grid.)

There are also three methods in **Util** that are already implemented, one for creating an array of **Jewel** objects from a string; one for a 2D array of **Jewel** objects from an array of strings; and one for converting a 2D array of **Jewel** objects to a printable string. This is intended to make your life easier when writing test cases for your code.

Overview of the JewelFactory class

The purpose of a **JewelFactory** makes **Jewel** objects. (Duh.) The key method is **generate()**, which returns a randomly generated new **Jewel** whose type (that is, its int value) is within a specified range. The range of values is determined at construction. You'll need to implement the constructors and the **generate** method (which are easy) pretty early on, since **generate** is required in the **Game** class to implement **shiftColumnDown** and **fillAll**.

In addition, the **JewelFactory** class specifies a method **createGrid**. This is harder, but is not needed for implementing the rest of the project except for the **Game** constructor that takes a width and height. In this constructor, only the width and height of the grid are given, and the jewels are randomly generated by the **createGrid** method. (The other constructor, that takes a string, is not dependent on the **createGrid** method.) The challenge is that the initial grid should not include any runs. For example, if you are iterating across the rows and generating a value for the cell at (row, col), and if you find that the two cells to the left both have some value v1 and the two cells above both have some value v2, then for (row, col) you need to generate a random value from the set of numbers between 0 and the max, *excluding* v1 and v2.

Overview of the Game class

This class encapsulates all the game state, primarily consisting of

- a 2D array of **`Jewel`** objects,
- an instance of **`JewelFactory`**, and
- a score.

(*Tip*: you should only need three instance variables!)

From the high-level perspective of a client using the **`Game`** class (such as a user interface) the four most fundamental operations are the following:

`boolean select(GridPosition c0, GridPosition c1)`
attempts to exchange a pair of cells

`ArrayList<GridPosition> findAndMarkAllRuns()`
finds all runs, sets all cells to null that are part of a run, and updates the score

`ArrayList<GridPosition> shiftColumnDown(int col)`
shifts the **`Jewel`**s in non-null cells down to fill null cells below

`ArrayList<GridPosition> fillAll()`
assigns new **`Jewel`**s to the null cells

Normally, these operations would normally be invoked in the order above, and the last three steps would be called repeatedly by the client until **`findAndMarkAllRuns`** does not find any runs (returns an empty list). However, the methods should work as specified even if invoked in a different order. (Exception: methods that find runs, including the first two operations above, are not expected to work if the grid contains any null cells.)

Notice that several of the methods have a return value of type **`ArrayList<GridPosition>`**. This is a list of cells of the grid that are affected in some way by the method, and the purpose of the list is to enable a user interface to display what is happening in the game. See the Javadoc for details about these lists. The game also includes basic accessors such as **`getWidth()`**, **`getHeight()`**, **`getJewel(int row, int col)`**, and **`getScore()`** that would be needed by any user interface to display the game.

In addition to the four principal methods above, there are some methods that are specified to help decompose the overall implementation into simpler parts that can be tested more easily. These methods could have been made private, since they are not expected to be called by clients, but we have made them public here to simplify testing. These include the following methods.

`Jewel[] getRow(int row)`
returns a copy of the specified row of the grid

`Jewel[] getCol(int col)`
returns a copy of the specified column of the grid

`void setRow(int row, Jewel[] arr)`
copies the given array values into the specified row of the grid

`void setRow(int row, Jewel[] arr)`
copies the given array values into the specified row of the grid

These methods make it easy to use the two methods of the **`Util`** class, which take a one-dimensional array, to operate on an individual row or column in the game grid.

The problem of finding runs is decomposed into two simpler methods, both of which should be implemented using the **`Util`** method **`findRuns`**:

`ArrayList<GridPosition> findHorizontalRuns(int row)`
finds all runs in the given row

`ArrayList<GridPosition> findVerticalRuns(int col)`
finds all runs in the given column

These two methods are accessors and do not modify the grid nor the score. Obviously, these can be used to implement **`findAndMarkAllRuns`**. (Note: to accumulate all **`GridPosition`**s into a final list to be returned, you might want to take advantage of the **`ArrayList`** method **`addAll`**.)

In addition, the two methods above are also useful to implement **`select`**. Note that if any runs are created by exchanging two cells, one of the runs must include at least one of two cells. Therefore one strategy for **`select`** is:

swap the jewels in the two cells
look for runs in the row and column for each of the two cells
if there are any runs, return true
otherwise swap back and return false

Importing the sample code

The sample code includes the **`hw3`**, **`api`**, and **`ui`** packages. It is distributed as a complete Eclipse project that you can import. **The code will not compile until you have created the Game and JewelFactory classes and stubbed in most of the methods.** Basic instructions:

1. Download the zip file to a location *outside* your workspace. You don't need to unzip it.
2. In Eclipse, go to File -> Import -> General -> Existing Projects into Workspace, click Next.
3. Click the radio button for "Select archive file".
4. Browse to the zip file you downloaded and click Finish.

Alternate procedure: If you have an older version of Java (below 8) or if for some reason you have problems with this process, or if the project does not build correctly, you can construct the project manually as follows:

1. Download the zip file to a location *outside* your workspace.
2. In Windows File Explorer right-click on the zip file and select Extract (Note: in OS X you can just double-click it to extract)
3. Create a new empty project in Eclipse
4. Browse to the src directory of the zip file contents
5. In the Package Explorer, navigate to the src folder of the new project.
6. Drag the **`hw3`**, **`ui`**, and **`api`** folders from Explorer/Finder into the **`src`** folder in Eclipse.

The GUI

The sample code includes a GUI in the **`ui`** package. The main method is in **`ui.GameMain`**. You select a pair of icons to swap by clicking dragging the mouse from one to the other. It won't work until you have started implementing the **`Game`** class; see the Getting Started section for more details

The GUI is built on the Java Swing libraries. This code is complex and specialized, and is somewhat outside the scope of the course. You are not expected to be able to read and understand it, though you might be interested in exploring how it works.

It's important to realize that the GUI *contains no actual logic or data for the game*. At all times, it simply displays the contents of your grid. It invokes the **`select`** method based on user mouse action. It invokes the **`findAndMarkAllRuns`** method if you return true from **`select`**. It then calls **`shiftColumnDown`** for each column, and finally calls **`fillAll`**. The latter three are called repeatedly until **`findAndMarkAllRuns`** returns an empty list. If the return values you provide from **`shiftColumnDown`** and **`fillAll`** are correct, the GUI will attempt to animate the "falling" motion of the icons.

So if something's not working, go back to your **`Game`** class and write some more unit tests!

All that the main class `GameMain` does is to initialize the components and start up the UI machinery. The class `GamePanel` contains most of the UI code and defines the "main" panel, and there is also a much simpler class `ScorePanel` that contains the display of the score. You can configure the game by editing the `create()` method of `GameMain`; see the comments there for examples. You can also edit the default colors here.

(*Optional reading*) If you are curious about what's going on, you might start by looking at the method `paintComponent` in `GamePanel`, where you can see the accessor methods such as `getJewel` being called to decide how to "draw" the panel. The most interesting part of any graphical UI is in the *callback* methods. These are the methods invoked when an event occurs, such as the user pressing a button or a timer firing. You could take a look at `MyMouseListener.mousePressed` and `MyMouseMotionListener.mouseReleased` in which you can see the call to your `select` method. (These are "inner classes" of `GamePanel`, a concept we have not seen yet, but it means they can access the `GamePanel`'s instance variables.)

If you are interested in learning more, there is a collection of simple Swing examples linked on Steve's website. See http://www.cs.iastate.edu/~smkautz/ and look under "Other Stuff". The absolute best comprehensive reference on Swing is the official tutorial from Oracle, http://docs.oracle.com/javase/tutorial/uiswing/TOC.html . A large proportion of other Swing tutorials found online are out-of-date and often wrong.

Testing and the SpecChecker

As always, you should try to work incrementally and write simple tests for your code as you develop it.

Do not rely on the GUI code for testing! Trying to test your code using a GUI is very slow, unreliable, and generally frustrating. *In particular, when we grade your work we are NOT going to run the GUI, we are going to verify that each method works according to its specification.*

Since test code that you write is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

We will provide a basic SpecChecker, but **it will not perform any functional tests of your code**. At this point in the course, you are expected to be able to read the specfications, ask questions when things require clarification, and write your own unit tests. There are extensive examples in the "Getting started" section.

The SpecChecker will verify the class names and packages, the public method names and return types, and the types of the parameters. If your class structure conforms to the spec, you should see a message similar to this in the console output:

```
x out of x tests pass.
```

This SpecChecker will also offer to create a zip file for you that will package up the two required classes. Remember that your instance variables should always be declared **`private`**, and if you want to add any additional "helper" methods that are not specified, they must be declared **`private`** as well.

Style and documentation

Special note: in this project you are writing some potentially complex loops that someone is going to have to read**. You are expected to write helpful internal comments for your code so that the reader can follow what you are trying to do.**

Internal comments are written **`//`**-style, not Javadoc-style, and are used inside of method bodies to explain *how* something works (while the Javadoc comments explain *what* a method does). If you had to think for a few minutes to figure out how to implement something, then add a comment to explain your strategy to the reader.

Internal comments always *precede* the code they describe, and are indented to the same level.

Roughly 15% of the points will be for documentation and code style. Here are some general requirements and guidelines:

- Use instance variables only for the "permanent" state of the object, use local variables for temporary calculations within methods.
    - You will lose points for having lots of unnecessary instance variables
    - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.
- Each class, method, constructor and instance variable, whether public or private, must have a meaningful and complete Javadoc comment. Class javadoc must include the **`@author`** tag, and method javadoc must include **`@param`** and **`@return`** tags as appropriate.
    - Try to state what each method does in your own words, but there is no rule against copying and pasting the descriptions from this document.
    - Run the javadoc tool and see what your documentation looks like! You do not have to turn in the generated html, but at least it provides some satisfaction :)

- All variable names must be meaningful (i.e., named for the value they store).

- Your code should not be producing console output. You may add `println` statements when debugging, but you need to remove them before submitting the code.

- If you have commented-out code, delete it before submitting. (If you want to save the commented out code to potentially review or restore later, just make a backup copy before you delete it.)

- Use a consistent style for indentation and formatting.
    - Note that you can set up Eclipse with the formatting style you prefer and then use Ctrl-Shift-F to format your code. To play with the formatting preferences, go to Window->Preferences->Java->Code Style->Formatter and click the New button to create your own "profile" for formatting.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder `assignment3`. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag `assignment3`. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in, and **for this assignment you are welcome to post and discuss test code**. (In the Piazza editor, use the button labeled "tt" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

Suggestions for getting started

At this point we expect that you basically know how to study the documentation for a class, write test cases, and develop your code incrementally to meet the specification. The suggestions below are mainly intended to help you think about how to test your code. The most important thing to remember is that you should not try to debug your code by running the GUI. Doing so is extremely slow and frustrating because of all the additional complexity of the GUI code itself. Write simple, focused test cases that you can easily understand and check. There are a number of examples below.

1. See the section "Importing the sample code". Remember that **the project will not compile** until you have created the **`Game`** and **`JewelFactory`** classes and provided stubs for the methods. Do that first and make sure you have no compile errors when you proceed. You can run the specchecker too.

2. Be sure you have read the earlier sections "Overview of the Util class", "Overview of the JewelFactory class" and "Overview of Game class". These sections provide a fairly detailed explanation of the design and you can start to see which parts of the code depend on other parts. You can start independently on all three classes, though eventually you'll get to a point where you need both the **`Util`** class and the **`JewelFactory`** to finally complete the **`Game`** class.

3. The first thing you'll need in **`Game`** is an instance variable to store the grid, which is a 2D array of **`Jewel`**. There are two **`Game`** constructors - one that takes a string array and one that takes a width and height. Start with the string array constructor; the other one depends on the **`JewelFactory`** method **`createGrid`**. To initialize your grid, all you need is to call the **`Util`** method **`createFromStringArray`**, which is already implemented, e.g., if **`myGrid`** is your grid instance variable, just write

```
myGrid = Util.createFromStringArray(descriptor);
```

See the section "Overview of the Game class" for suggestions for other instance variables you'll need.

4. Implement some of the one-liner methods involving the grid, such as **`getHeight`**, **`getWidth`**, **`getJewel`**, **`setJewel`**. The method **`toString`** is useful to have for testing and it is also a one-liner, since the real work is done already in the **`Util`** class, e.g., you can just write

```
return Util.convertToString(myGrid);
```

Write a few simple tests to make sure you can construct and examine a game and its grid, for example,

```
  public static void main(String[] args)
  {
    // descriptor for initial grid of a 3 x 4 game
    String[] desc =
      {
        "2 0 1 3",
        "1 0 1 3",
        "2 1 2 1"
      };

    JewelFactory generator = new JewelFactory(4);
    Game g = new Game(desc, generator);
    System.out.println(g.toString());

    System.out.println(g.getJewel(2, 1));  // expected 1
    g.setJewel(2, 1, new Jewel(3));
    System.out.println(g.getJewel(2, 1));  // expected 3
    g.setJewel(2,  1,  null);
    System.out.println(g.toString());

  }
```

This should produce the output,

```
 2 0 1 3
 1 0 1 3
 2 1 2 1
1
3
 2 0 1 3
 1 0 1 3
 2 * 2 1
```

> (Note that the **println** method will automatically invoke an object's **toString** method, if there is one, to convert it to a text representation. So in the last line we can just write **System.out.println(g)**.)

5. The **Game** methods **getRow**, **getCol**, **setRow**, and **setCol** are very straightforward loops and easy to test. For example, if **g** is the game created in the previous example, you should be able to write,

```
    Jewel[] arr = g.getCol(2);
    System.out.println(Arrays.toString(arr));
    System.out.println();
    Jewel[] testCol = Util.createFromString("5 6 7");
    g.setCol(1, testCol);
    System.out.println(g);
```

and see the output,

```
[1, 1, 2]

 2 5 1 3
 1 6 1 3
 2 7 2 1
```

6. You can work on **`JewelFactory`** independently; the constructors and the **`generate`** method are pretty simple. The **`createGrid`** method will be a bit tricky, but it is not needed until you implement the second constructor of **`Game`** (the one that takes a width and height).

7. You can also start developing and testing the methods of **`Util`**. As always, begin by doing lots of examples with pencil and paper, and carefully keep track of what you're doing. For example, consider the method **`findRuns`**. Before worrying about **`Jewel`** objects and returning an ArrayList and so on, just take an int array, say [4, 2, 2, 2, 4, 4, 3, 3, 3, 3, 2], and print some output to try to get a handle on the problem. As you scan the array from left to right, you might mentally keep a count of how many times you have seen the current value:

4	2	2	2	4	4	3	3	3	3	2
1	1	2	3	1	2	1	2	3	4	1

Could you iterate over the array and print that count at each cell (e.g. the small numbers below)? Could you print "foo" each time the count changes from 3 or more back down to 1? Could you print the index at which it occurs? Could you print the starting and ending indices of each run? Could you print all the indices for the cells making up the run?

Next, think about some concrete test cases for the real method. For example,

```
Jewel[] test = Util.createFromString("4 2 2 2 4 4 3 3 3 3 2");
ArrayList<Integer> result = Util.findRuns(test);
System.out.println(result); // expected [1, 2, 3, 6, 7, 8, 9]
```

> *Tip:* Remember that when you work with **`Jewel`** objects, although you can *print* them like integers, you have to use the **`.equals`** method to compare them, not **`==`**. Be careful in the case that the last element of a run is at the end of the array.

8. Once you have **`Util.findRuns`** working, you can combine it with **`getRow`** and **`getCol`** to implement **`findHorizontalRuns`** and **`findVerticalRuns`**. *Do not re-implement the algorithm for findRuns!* Make a few test cases, for example,

```
String[] desc =
  {
    "4 2 2 2 4 4 3 3 3 3 2",
    "1 2 3 4 5 1 2 3 4 5 1"
  };
```

```
JewelFactory generator = new JewelFactory(6);
Game g = new Game(desc, generator);
ArrayList<GridPosition> result = g.findHorizontalRuns(0);
System.out.println(result);
```

The expected output is a list of **`GridPosition`** objects, one for each cell that is part of a run in the given row (or column), listed left to right (or top to bottom):

```
[[(0,1) 2], [(0,2) 2], [(0,3) 2], [(0,6) 3], [(0,7) 3], [(0,8) 3], [(0,9) 3]]
```

The string representation of a **`GridPosition`** object is "[(*row*, *column*) *jewel*]"

9. Once you have **`findHorizontalRuns`** and **`findVerticalRuns`**, you can implement **`select`**. The last paragraph of the section "Overview of the Game class" gives some suggestions for implementation. How would you test it? Take an example grid, say

```
3 3 2 1 0
3 3 0 0 1
2 2 3 2 1
0 2 3 1 0
```

Exchanging the value at row 1, column 1 with the value at row 1, column 2 should result in a run in column 2. The grid should be modified, and the select method should return true:

```
String[] desc =
  {
    "3 3 2 1 0",
    "3 3 0 0 1",
    "2 2 3 2 1",
    "0 2 3 1 0"
  };
JewelFactory generator = new JewelFactory(6);
Game g = new Game(desc, generator);
GridPosition[] cells = {new GridPosition(1, 1, null),
                        new GridPosition(1, 2, null)};
boolean ret = g.select(cells);
System.out.println(ret);
System.out.println(g);
```

The expected output is:

```
true
 3 3 2 1 0
 3 0 3 0 1
 2 2 3 2 1
 0 2 3 1 0
```

10. The method **findAndMarkAllRuns** can be implemented from **findHorizontalRuns** and **findVerticalRuns**. You might want to take advantage of the ArrayList method **addAll** to accumulate the resulting **GridPosition**s into one big ArrayList. The "marking" part means that each cell that you find as part of a run needs to be set to null in the grid. (When you print out the grid for testing with **toString**, the null cells are indicated with the '*****' character.) Don't forget about the score too.

11. The method **fillAll** can be written anytime and is straightforward. If you don't have the **generate()** method of your **JewelFactory** working yet, just leave the stub that always returns zero. You can test **fillAll** by creating a game and nulling out some cells:

```
String[] desc =
  {
    "2 4 1 3",
    "1 4 1 3",
    "2 1 2 1"
  };

Game g = new Game(desc, new JewelFactory(5));
g.setJewel(0, 1, null);
g.setJewel(1, 1, null);
g.setJewel(0, 3, null);
g.setJewel(2, 0, null);
System.out.println(g);
ArrayList<GridPosition> result = g.fillAll();
System.out.println(g);
System.out.println(result);
```

This produces output like the following (in this example, the **JewelFactory** has produced four jewels with type 0):

```
2 * 1 *
1 * 1 3
* 1 2 1

2 0 1 0
1 0 1 3
0 1 2 1

[[(0,1) 0], [(0,3) 0], [(1,1) 0], [(2,0) 0]]
```

12. To go further you'll need to implement **Util.shiftNonNullElements**. There are many reasonable ways to do this, and you have seen similar examples in class. One idea is to make a temporary ArrayList consisting of the non-null elements; then copy them back into the end of the array. Start with some test cases so you have a clear before-and-after picture of what you're trying to do. For example:

```
Jewel[] test = Util.createFromString("0 1 2 3 4 5 6 7");
test[2] = null;
test[5] = null;
System.out.println(Arrays.toString(test));
ArrayList<Integer> result = Util.shiftNonNullElements(test);
System.out.println(Arrays.toString(test));
System.out.println(result);
```

This should produce the output,

```
[0, 1, null, 3, 4, null, 6, 7]
[null, null, 0, 1, 3, 4, 6, 7]
[2, 3, 4, 5]
```

Notice that the elements actually moved are the 0, the 1, the 3, and the 4, and the returned ArrayList contains the *indices* of those elements in the *modified* array.

13. The method **`shiftColumnDown`** is based on **`Util.shiftNonNullElements`**, so you could do that next. *Do not reimplement the algorithm for shiftNonNulElements!* The use of the returned list of integers may be confusing, so write an example to make sure you know what you want it to do. Here is one where we put some nulls in column 1 and then shift it down.

```
String[] desc =
{
  "0 0 0",
  "0 1 0",
  "0 2 0",
  "0 3 0",
  "0 4 0",
  "0 5 0",
  "0 6 0",
  "0 7 0"
};


Game g = new Game(desc, new JewelFactory(8));
g.setJewel(2, 1, null);
g.setJewel(5, 1, null);
System.out.println(g);
ArrayList<GridPosition> result = g.shiftColumnDown(1);
System.out.println(g);
System.out.println(result);
```

This should produce the output,

```
 0 0 0
 0 1 0
 0 * 0
 0 3 0
 0 4 0
 0 * 0
 0 6 0
 0 7 0

 0 * 0
 0 * 0
 0 0 0
 0 1 0
 0 3 0
 0 4 0
 0 6 0
 0 7 0

[[(2,1) 0], [(3,1) 1], [(4,1) 3], [(5,1) 4]]
```

14. The second Game constructor - the one that takes a width and height - depends on the method **`createGrid`** of **`JewelFactory`**. You'll need to come up with a way to fill up the 2D array with randomly generated values while preventing it from having runs. There are many ways to go about this but what you must **NOT** do is something like:

while(there are no runs)
fill all the cells with random values

That could take millions of iterations and is extremely inefficient. Instead think about how, when assigning a jewel for a particular cell, you can limit the random values from which you are selecting a number for that cell. For example, if your max is 6 and you know you can't put a 2 or a 3 in the cell, can you select a random value from the set [0, 1, 4, 5] ? One easy strategy is to make an ArrayList of those values and then select a random index. So if you iterate over the grid top to bottom and left to right, then at each cell, look at its two neighbors to the left - if they both have value x, then avoid generating another x; look at the two neighbors above - if they both have value y, then avoid generating another y.

15. Once you get through step 13 you should be able to run the GUI and play the game. Edit the main method in **`GameMain`** to change the initial grid. If you have both Game constructors implemented, you can make a game that starts with random values (see the commented code in **`GameMain`**.

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Homework page on Blackboard, before the submission link will be visible to you.

Please submit, on Blackboard, the zip file that is created by the second SpecChecker. The file will be named **`SUBMIT_THIS_hw3.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`hw3`**, which in turn contains the three files **`Game.java`**, **`JewelFactory.java`**, and **`Util.java`**.

Please LOOK at the zip file you upload and make sure it is the right one!

Submit the zip file to Blackboard using the Assignment 3 submission link and **verify that your submission was successful by checking your submission history page**. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links."

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw3**, which in turn should contain the three required files. Make sure all files have the extension **`.java`**, NOT **`.class`**. You can accomplish this easily by zipping up the **src** directory of your project. The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and **not** a third-party installation of WinRAR, 7-zip, or Winzip.