Com S 227
Fall 2022
Assignment 3
300 points

Due Date: Friday, November 4, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Nov 3)

10% penalty for submitting 1 day late (by 11:59 pm Nov 5)

No submissions accepted after Nov 5, 11:59 pm

This assignment is to be done on your own. See the Academic Integrity policy in the syllabus, for details.
You will not be able to submit your work unless you have completed the *Academic Dishonesty policy questionnaire* on the Assignments page on Canvas. 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.

Please start the assignment as soon as possible and get your questions answered right away. It is physically impossible for the staff to provide individual help to everyone the afternoon that the assignment is due!

This is a "regular" assignment so we are going to read your code. Your score will be based partly (about a third) on the specchecker's functional tests and partly on the grader's assessment of the quality of your code. See the "More about grading" section.

Contents

Overview

You will be writing the logic for a version of a video puzzle game that is primarily based on the game “Lyne”, though it also shares elements with “Flow Free". You'll get lots of practice working with arrays, ArrayLists, and files. 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 two classes **`hw3.LinesGame`** and **`hw3.Util`**.

The game is typically viewed as a grid in which pairs of cells, which we call the *endpoints*, are marked with the same color. A mouse or pointing device is then used to draw a line between the matching pairs. However, there are restrictions: the non-endpoint cells are normally marked with colors too, and the line between the endpoints can only pass through intermediate cells with the same color. The lines cannot cross, except through specially designated calls we call “crossings”.

For example, here is a simple grid as viewed in the sample GUI. The left image is the initial state. The large circles are endpoints, and the small circles are the “middle” cells. The grey circles are crossings, and the number shows the maximum number of times the cell can be

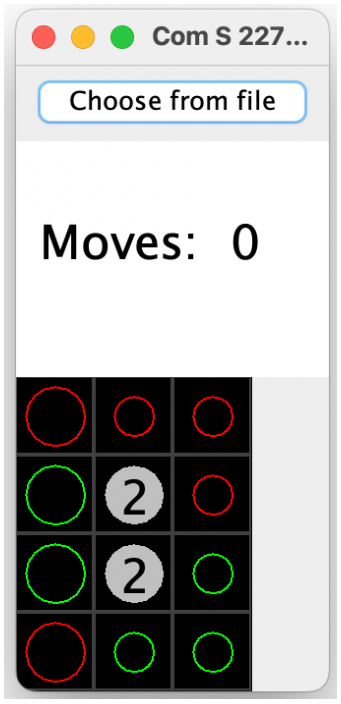

Figure 1 - game state before any moves

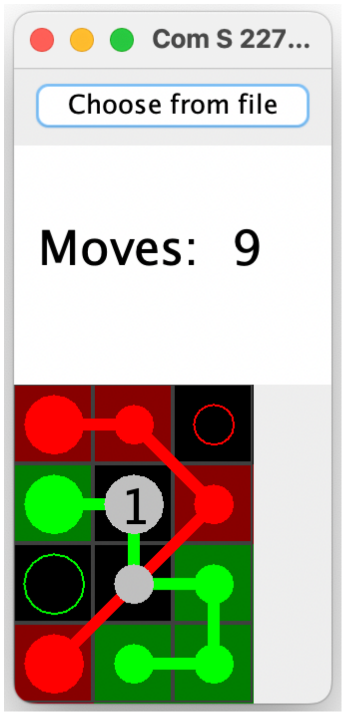

Figure 2 - game state after some moves

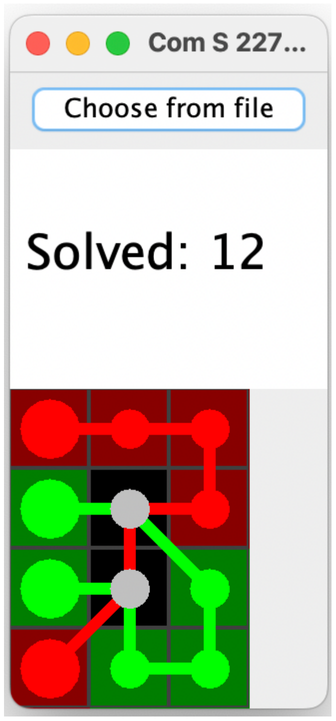

Figure 3 - game state on completion

crossed by a line. In the center figure, the numbers on the crossings are reduced by how many times the cell has already been crossed. The small grey circle indicates a crossing whose count has reached its maximum. In this example, the player has made some mistakes: the red line is "complete", meaning that its endpoints are connected, but one of the red middle cells has been missed; also, there is no way the player can win since the green line can't cross the red line to get to the second green endpoint. In addition, to be considered a solution, all the middle cells have to be included and all crossings have to be fully used. On the right, the game is complete, meaning that all the lines are complete, all non-crossing cells are crossed exactly once, and each crossing cell has been crossed the maximum number of times.

You can see a short video of the game being played here: https://www.lynegame.com/
And here are some examples with harder puzzles:
https://www.youtube.com/watch?v=Hn8QVWYbqmI

Implementation summary

The abstractions we will use for representing the game state are three simple classes called **`Location`**, **`GridCell`** and **`Line`**, located in the **`api`** package of the sample code. (*These three classes are fully implemented and you should not modify them.*)

A **`Location`** just represents a row and column pair.

A **`GridCell`** is a cell in the 2d grid for the game. There are four types, indicated by the enum constants in **`CellType`**.

- OPEN – has no id, a line with any id can go through once
- MIDDLE – has an id, and only a line with matching id can go through once
- ENDPOINT – has an id, can only be the start or end of a line
- CROSSING – has no id, a line with any id can go through, and the number of lines is limited by its max count

Each type has a *count* and a *max count*. For the first three types, the max count is always 1, meaning that only one line can go through that cell. For a crossing, the max count is set in its constructor. Whenever a line goes through a cell, the count is incremented by 1; likewise, when a line through a cell is deleted (which happens when an endpoint is selected) the count is decremented by 1. The id values are, in principle, arbitrary non-negative integers, but in practice will most likely be numbers from 0 through 12 representing possible colors. An open cell or a crossing cell always has an id of -1 (indicating that a line of any color can go through it).

A `Line` encapsulates a pair of endpoints and a sequence of `Locations` in the grid that may eventually connect the endpoints; specifically, it has

- an integer id
- two `Location` objects representing the matching endpoints to be connected
- an `ArrayList` of `Location` objects representing the sequence of cells in the line, initially empty

The two endpoints are always present within the `Line` object, but are not necessarily in its list of locations. That list is initially empty, and may grow (or become empty again) based on user actions. The two endpoints are in no particular order. The id essentially represents the color for the line and is always the same as the id for its endpoints. A line can, in general, only pass through cells in the grid that have a matching id (the exceptions are the crossing cells and the "open" cells that have no id).

Game state

Overall the complete state of the game consists of

- a 2d array of `GridCell` objects,
- an ArrayList of `Line` objects,
- a designated "current" line (possibly null), and
- a counter for moves made

Intuitively, the "current line" corresponds to the line currently being manipulated by the user (e.g. by moving a mouse). The "current cell" is just the last location in the current line's list of positions.

Basic operations

The basic operations on the game are *startLine(row, col)*, *endLine()*, and *addCell(row, col)*.

The *startLine* operation sets the current line. When using a GUI, it is typically invoked when the mouse is *pressed*. The operation does nothing if the current line is already non-null. There are restrictions on which cells can be selected by a *startLine* operation:

- Any endpoint can be selected. Selecting an endpoint clears the line associated with that endpoint and then adds the selected cell to the line, which then becomes the current line.
- A non-endpoint can be selected only if it is the last cell in a line and it is not a crossing. That line then becomes the current line.

The *endLine* operation basically just sets the current line to null. When using a GUI, it is typically invoked when the mouse is *released*. (While the current line is null, the mouse can be moved without triggering any *addCell* operations.)

The *addCell* operation attempts to add a new location onto the current line. When using a GUI, it is typically invoked when the mouse is *dragged*. The restrictions are:

1. The current line is non-null
2. The current line is not already *connected* (i.e., doesn't include both endpoints)
3. The given location is adjacent to the current cell (horizontally, vertically, or diagonally)
4. The count for the new cell is less than its max count
5. If the new cell is a middle or endpoint cell, then its id matches the id for the current line
6. Adding the new cell will not re-trace a segment of any existing line
7. Adding the new cell to the line would not cross any existing line

If the conditions are met, a new location with the given row/column is added to the current line, and the count for the new cell is incremented.

About OPEN cells

The `GridCell` class and the accompanying enum type `CellType` allow a cell to be designated as OPEN. This means that it can be crossed by a line only once, like a MIDDLE cell, but the id is not restricted. A game in which all non-endpoint cells are open is similar to the classic game "Flow Free". An example might look like this in a partially completed state:

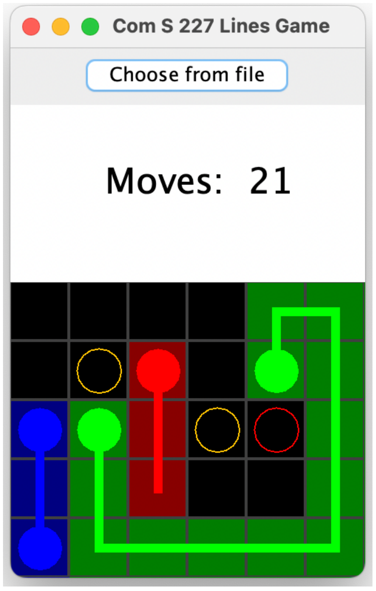

Figure 4 - a game that only has endpoints and open cells

The Util class

You will also be implementing a class called `Util`. This is a "utility" class containing a few static methods:
`createLinesFromGrid` – given a 2d array of `GridCell`, construct a list of `Line` objects, one for each matching pair of endpoints
`readFile` – read a file containing string descriptors for initial game states, and construct an array of `LinesGame` objects from it
`checkForLineSegment` – given a list of `Line`s and two locations, determine whether any existing line already connects the two locations
`checkForPotentialCrossing` – given a list of `Line`s and two locations, determine whether any existing line would be crossed by connecting the two locations

Specification

The specification for this assignment includes this pdf, the online Javadoc, and any "official" clarifications announced on Canvas.

The sample code includes a fully documented skeleton of `LinesGame.java` and `Util.java.` Read the javadoc comments for details about the precise behavior of methods. You can also find all the documentation in html form on Canvas.

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

String representations and the StringUtil class

It is convenient to represent an initial game state with an array of strings in which all strings are the same length. Such an array can be interpreted as a grid. Here is an example showing a string representation of the initial state of the game in Figure 1:

```
String[] testgrid3 = {
  "Rrr",
  "G2r",
  "G2g",
  "Rgg"
};
```

Capital letters represent endpoints, and corresponding lowercase letters indicate middle cells with the same id. The numbers represent crossing cells and the numeric value is the max count. A dash is an open cell. Here is a string representation of the initial state of the game in Figure 4:

```
String[] testflow0 = {
  "------",
  "-OR-G-",
  "BG-OR-",
  "------",
  "B-----"
};
```

The correspondence between letters and cell id values is in principle arbitrary, but it is convenient to adopt a consistent convention. In our case, the convention is that *the id value for a given letter is its index in the array:* `api.StringUtil.COLOR_CODES`. (The letters are chosen so as to be mnemonic for the color array `ui.GamePanel.COLORS` used in the in the sample GUI code, e.g., 'R' means RED, 'G' means GREEN, etc.)

The StringUtil class

As an aid in constructing and debugging games, there is a utility class provided for you called `api.StringUtil` that includes the following methods:

`createGridFromStringArray` – from a string array such as "testgrid3" above, returns a 2d array of `GridCell`
`getIdForCharacter` – from one of the possible characters in a string representation of a grid, returns its id (index in the COLOR_CODES array)
`originalGridToString` – given a 2d array of `GridCell`, returns a string representation of the *original* game state that it represents
`currentGridToString` – given a 2d array of `GridCell`, returns a string representation of the *current* game state that it represents
`allLinesToString` – given a list of `Line` objects, returns a string representation of all of them

Note that the skeleton for the `LinesGame` class also includes an implemented `toString` method that uses the above to return a string representation of the complete game state, i.e., the original grid, the current state, and the lines. The classes `Position` and `Line` each include a `toString` method, so they can be easily printed using System.out.println.

Importing the sample code

The sample code includes a complete skeleton of the two classes you are writing in the **`hw3`** package along with supporting code in the **`api`** and **`ui`** packages. It is distributed as a complete Eclipse project that you can import. It should compile without errors out of the box. *However the GUI will not run correctly until you have implemented the basic functionality of the game.* Basic instructions for importing:

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 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. Unzip the zip file containing the sample code.
2. In Windows File Explorer or OS X Finder, browse to the src directory of the zip file contents
3. Create a new empty project in Eclipse. **Be sure to UNCHECK the box for "Create module-info"**
4. In the Package Explorer, navigate to the src folder of the new project.
5. 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 can try running it, and you'll see an initial window, but **until you start implementing the required classes you'll just get errors** (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 information returned by your **`getLines()`** and getCell() methods, using the values of your **`getRows()`** and **`getColumns()`** methods for the size. It invokes the **`startLine`** method when the user presses the mouse; it invokes the **`endLine`** method when the user releases the mouse, and invokes the **`addCell`** method when the user drags the mouse. It ends the game when your **`isComplete`** method returns true.

So if something's not working, go back to your **`LinesGame`** 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 setting editing the **`create()`** method of **`GameMain`**; see the comments there for examples.

You may also notice the button "Choose from file" at the top of the UI. When your Util class is done and debugged, you can try it. It should bring up a file dialog and allow you to select a file. It will then attempt to read your file and obtain the list of **`LinesGame`** objects returned by the **`Util.readFile`** method. It then brings up a dialog with a drop-down list allowing you to select one of the games that was found in the file.

> (*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 **`getCell`** and **`getAllLines`** 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.mouseDragged`** in which you can see the calls to your **`startLine`** and **`addCell`** methods. (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. 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 tests for your code as you develop it.

Do not rely on the UI code for testing! Trying to test your code using a UI is very slow, unreliable, and generally frustrating. The code for the UI itself is more complex than the code you are implementing, and it is not guaranteed to be free of bugs. ***In particular, when we grade your work we are NOT going to run the UI, we are going to test that each method works according to its specification.***

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 specifications, ask questions when things require clarification, and write your own unit tests.

Since the test code 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.**

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.

Suggestions for getting started

*Remember to work **incrementally** and test new features as you implement them. At this point in the course, we expect that you can study this document and read the javadoc and construct usage examples or simple test cases to guide your implementation. Here are some suggestions for how you might approach the process of testing. Note that the code snippets below can be found in the SimpleTest class in the default package of the sample code.*

1. Take a few minutes and familiarize yourself with the classes in the **`api`** package. You will need to use them to get started at all.

2. To a great extent you can work independently on **`LinesGame`** and **`Util`**, except that:

- The one-argument constructor for **`LinesGame`** depends on **`Util.createLinesFromGrid`**
- There are some details of **`addCell`** that depend on **`Util.checkForLineSegment`** and **`Util.checkForPotentialCrossing`**, but you can get most of it done without those two methods
- The method **`Util.readFile`** depends on having the one-argument constructor for **`LinesGame`** working

3. There are two constructors for the **`LinesGame`** class. It is easiest to start with the two-argument constructor, which is harder to use but simple to implement. In order to use it, we have

to provide a 2d array of **`GridCell`** for the game grid, and a list of **`Line`**s that is derived from the grid. To make the 2d array, we can use the utility method **`StringUtil.createGridFromStringArray`**. For example,

```
String[] test = {
  "GRrR",
  "ggGY",
  "Yyyy"
};
GridCell[][] grid = StringUtil.createGridFromStringArray(test);
```

To examine the grid, we can use one of the XXXtoString methods of **`StringUtil`**:

```
System.out.println(StringUtil.originalGridToString(grid));
```

and we should see:

```
G R r R
g g G Y
Y y y y
```

To make a game, we also need a list of **`Line`** objects, one for each pair of endpoints. The most convenient way to do this would be with **`Util.createLinesFromGrid`**, but you can also do it manually, in order to start testing your game class before implementing that method. Look at the COLOR_CODES array in **`StringUtil`** and note that 'G' has index 1, 'R' has index 0, and 'Y' has index 4. Those should be the ids for the three corresponding lines. Carefully identify the (row, column) for each endpoint and construct a location for each one to construct the **`Line`** objects. Note there is no particular order among the lines in the ArrayList.

```
ArrayList<Line> lines = new ArrayList<>();

// endpoints for 'G' which corresponds to id 1
lines.add(new Line(1, new Location(0, 0), new Location(1, 2)));

// endpoints for 'R' which corresponds to id 0
lines.add(new Line(0, new Location(0, 1), new Location(0, 3)));

// endpoints for 'Y' which corresponds to id 4
lines.add(new Line(4, new Location(2, 0), new Location(1, 3)));
```

We can also print out the list of lines with another **`StringUtil`** method:

```
System.out.println(StringUtil.allLinesToString(lines));
```

which should produce the output:

```
id 1(G): {(0, 0), (1, 2)} []
id 0(R): {(0, 1), (0, 3)} []
id 4(Y): {(2, 0), (1, 3)} []
```

Each string of output shows the two endpoints in braces, followed by the list of locations (initially empty). If the line endpoints appear to match the grid, we can construct a game:

```
    LinesGame game = new LinesGame(grid, lines);
```

The grid and the lines can initially be stored in the game object without modification and examined for testing purposes using some simple accessor methods that you can easily implement:

```
    System.out.println(grid == game.getGrid());      // expected true
    System.out.println(lines == game.getAllLines()); // expected true
```

Once those two accessors are working you can also print out the game using its own **`toString`** method (which is already implemented in the skeleton code).

```
    System.out.println(game);
```

which produces the output,

```
-----
G R r R
g g G Y
Y y y y
-----
. . . .
. . . .
. . . .
-----
id 1(G): {(0, 0), (1, 2)} []
id 0(R): {(0, 1), (0, 3)} []
id 4(Y): {(2, 0), (1, 3)} []
Current line: null
```

The first block above is the “original” state of the grid, which does not change. The second block is the “current” game state, which would show which cells are currently crossed by lines. Since we haven’t made any moves yet, there is nothing to see.

4. At this point you might start filling out the easy accessor methods and think about what additional instance variables you might need (see the section "Game state" on page 4). These methods are all very straightforward. (The **`Line`** method **`getLast`** is useful for implementing the last one.)

```
getWidth
getHeight
getCell
getCurrentLine
getCurrentId
getMoveCount
getCurrentLocation
```

5. Starting and ending a line is initially pretty simple: **`startLine`** makes the line associated with an endpoint into the "current" line, and **`endLine`** sets the current line back to null. Here's a simple usage example:

```
    game.startLine(1, 3); // (1, 3) is an endpoint for id 4 (Y)
    Line line = game.getCurrentLine();
    System.out.println(line);
```

and we expect to see:

```
id 4(Y): {(2, 0), (1, 3)} [(1, 3)]
```

Note that the starting endpoint (1, 3) has been added to the list for line 4. A second effect should be that the cell at (1, 3) is now considered "occupied", that is, its count is incremented from 0 to 1.

```
   GridCell gc = game.getCell(1, 3);
   System.out.println(gc.getCount()); // expected 1
```

and if we look at the current grid, it will now show cell (1, 3) as occupied:

```
    System.out.println(StringUtil.currentGridToString(game.getGrid()));
```

expected:

```
. . . .
. . . Y
. . . .
```

Looking at the javadoc, you will see that there are several additional things to do in **`startLine`**, but this will do for the moment.

6. The method **`addCell`** will eventually be the most complex, but you can start out simply. It should initially check that there is a current line, that the given (row, col) is adjacent to the current cell (last cell in the current line), that the cell there has an id that matches the current line's id, and that the cell's count is less than its max count. (The id -1 is considered to "match" any id.) If these conditions are met, the cell's count is incremented and the (row, col) is added to the line. Continuing the previous example, we could write:

```
    game.addCell(2, 3);
    game.addCell(2, 2);
    System.out.println(game);
```

which should produce output,

```
-----
G R r R
g g G Y
Y y y y
-----
. . . .
. . . Y
. . y y
-----
id 1(G): {(0, 0), (1, 2)} []
id 0(R): {(0, 1), (0, 3)} []
id 4(Y): {(2, 0), (1, 3)} [(1, 3), (2, 3), (2, 2)]
Current line: 4
```

7. At this point you should be able to study the javadoc and start adding and testing additional features. You can use the simple testing strategy above to start making up new tests or usage examples, for instance:

- *calling startLine when the current line is non-null has no effect*
- *after endLine, the current line is null*
- *calling startLine on an endpoint should clear the existing locations in that line and decrement the counts of the previously occupied cells*
- *calling addCell when the current line is null has no effect*
- *calling addCell for a non-adjacent (row, col) has no effect*
- *calling addCell for a cell with non-matching id should have no effect*
- *addCell should always treat open cells or crossings as having a matching id*
- *calling addCell for a cell whose count is at its maximum should have no effect*

and so on.

More about grading

This is a "regular" assignment so we are going to read your code. Your score will be based partly (about a third) on the specchecker's functional tests and partly on the TA's assessment of the quality of your code. This means you can get partial credit even if you have errors, and it also means that even if you pass all the specchecker tests you can still lose points. Are you doing things in a simple and direct way that makes sense? Are you defining redundant instance variables? Some specific criteria that are important for this assignment are:

- 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 unnecessary instance variables
  - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.
- You should not have a lot of redundant code. If a set of actions is repeated in multiple places, create a helper method.

See the "Style and documentation" section below for additional guidelines.

Style and documentation

Roughly 15% of the points will be for documentation and code style. Here are some general requirements and guidelines in addition to those noted above.

- **Each class, method, constructor and instance variable, whether public or private, must have a meaningful javadoc comment**. The javadoc for the class itself can be very brief, but must include the **`@author`** tag. The javadoc for methods must include **`@param`** and **`@return`** tags as appropriate.
  - Try to briefly state what each method does in your own words. However, there is no rule against copying the descriptions from the online documentation.
  - 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.
- Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. (A good

rule of thumb is: if you had to think for a few minutes to figure out how something works, you should probably include an internal comment explaining how it works.)
    - Internal comments always *precede* the code they describe and are indented to the same level.
- 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 **`hw3`**. 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 **`hw3`**. *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. (In the Piazza editor, use the button labeled "pre" 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 announcements from the instructors on Canvas 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 Canvas. (We promise that no official clarifications will be posted within 24 hours of the due date.)

What to turn in

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

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw3.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw3`**, which in turn contains two files, **`LinesGame.java`** and **`Util.java`**. Please **LOOK** at the file you upload and make sure it is the right one!

Your submission does not include the **`api`** or **`ui`** packages.

Submit the zip file to Canvas using the Assignment 3 submission link and **VERIFY** that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO", link #11 on our Canvas front page.

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 files **LinesGame.java** and **Util.java**. You can accomplish this by zipping up the **src** directory of your project as described in the submission HOWTO document. **Do not zip up the entire 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.