Com S 227
Fall 2020
Assignment 3
250 points

Due Date: Tuesday, October 27, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Oct 26)

10% penalty for submitting 1 day late (by 11:59 pm Oct 28)

No submissions accepted after October 28, 11:59 pm

(Remember that Exam 1 is Thursday, October 29. You should really really really try to get this done Tuesday at the latest!)

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 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!

Introduction

The purpose of this assignment is to give you some practice writing loops, using arrays and lists, and most importantly to get some experience putting together a working application involving several interacting Java classes.

There are two classes for you to complete: `Pearls` and `PearlUtil`. As always, your primary responsibility is to implement these classes according to the specification and test them carefully. The class `PearlUtil` includes *working* implementations of the same algorithms you worked on

in miniassignment 2; the task you need to completee in **`PearlUtil`** is to add code to update the **`MoveRecord`** objects, which are explained below.

These two classes can be used, along with some other components, to create a simplified implementation of a lovely video puzzle game called "Quell." If you are not familiar with the game, do not worry, it is not complicated. One easy way to get familiar with the basic gameplay is to watch one of the YouTube videos about it, e.g.,

https://www.youtube.com/watch?v=KXl13okWA8Q

The image above is a screenshot from the original game. There is a grid consisting of various items, which we will call *cells*. There is also a *player*, pictured above as the blue "raindrop". Other types of cells seen above include the small round *pearls*, the darker square which is a *movable block*, the red pointy *spikes*, and the gold rings which are *portals*.

The player can be directed to move left, right, up, or down, and it will continue to move in a given direction until it reaches a boundary such as a wall. The goal is to collect all the pearls without getting stuck by the spikes. (It is ok to bump into the spikes on the non-pointy side.) The portals occur in pairs - going into one portal, the player will emerge, in the same direction, from its companion portal. The movable blocks can be pushed by the player. In addition, there are two kinds of movable blocks that we refer to as having "positive" and "negative" parity; when two movable blocks of opposite parity are pushed together, they vanish (according to the algorithm described in **`PearlUtil.moveBlocks`**). Not illustrated in the picture above is the fact that it is not actually necessary to have walls on all sides - if the player goes off the edge of the grid, it wraps around to the other side. Also not seen is one additional type of cell, called a *gate*. The player can pass through a gate, but only once; after that it is closed (and acts like a wall).

Your task in implementing **`Pearls`** and completing **`PearlUtil`** is to provide all the backend logic for the state of the game. Basically, **`Pearls`** encapsulates a 2D array representing the grid, the

player's current position, the current score (count of pearls collected) and the number of moves so far.

To have some fun with it, we will provide a couple of user interfaces: a very clunky console interface and a graphical interface that is capable of animating the motion of the player and movable blocks. You'll find that it's essential, however, to test your methods independently of the UIs. More details on the UIs below.

The sample code includes a partial skeleton for **`Pearls`** in the package **`hw3`**. A few of the methods of the **`Pearls`** class are already implemented, and there is a partial implementation of a constructor for initializing a 2D array of **`Cell`** objects. The additional code is in the packages **`ui`** and **`api`**. The **`ui`** package is the code for the GUI, described in more detail in a later section. The **`api`** package contains some relatively boring types for representing data in the game.

You should not modify any of the code in the **`api`** package.

Specification

The specification for this assignment includes

- this pdf,
- any "official" clarifications posted on Piazza, and
- the online javadoc

Basic types

The grid is represented as a 2D array of **`Cell`** objects. Each cell encapsulates a **`State`**, which is an **`enum`** type with the following values:

```
EMPTY,
WALL,
PEARL,
OPEN_GATE,
CLOSED_GATE,
MOVABLE_POS,
MOVABLE_NEG,
SPIKES_LEFT,
SPIKES_RIGHT,
SPIKES_DOWN,
SPIKES_UP,
SPIKES_ALL,
PORTAL;
```

A **`Cell`** object also includes a flag indicating whether the player is present in the cell, and also two integers, a *row offset* and *column offset*, that are used only for portals to store the relative location of the companion portal. The **`State`** class has a few static methods that may be useful when working with states.

There is also an **`enum`** type **`Direction`** with the four values **`LEFT, RIGHT, UP, DOWN`**.

The **`api`** package includes a somewhat bulky utility class called **`StringUtil`** that has a number of methods for representing cell states as strings and for converting back and forth from strings to **`Cell`** arrays. You'll find these methods are extremely useful for testing. (The string methods previously located in **`mini2.State`** and **`mini2.PearlUtil`** have been moved into **`StringUtil`**.)

*However, **`StringUtil`** should be used ONLY for testing. You should not be using **`StringUtil`** in your **`PearlUtil`** or **`Pearls`** code; use the methods of **`Cell`** and **`State`**. The only dependence on **`StringUtil`** should be in the (given) **`Pearls`** constructor.*

If you look at the static char array **`StringUtil.TEXT`**, you can see the conventions we will use for representing the game grid in string format. For example, the grid shown in the sample screenshot earlier would look like this in our textual format, where the **`'$'`** character is used to indicate the player location.

```
. # # # # # .
# . . . . # #
# $ . o . @ #
# . . . . # #
# . o . . . #
# . . + . . #
# ^ . # # @ #
. # # # # # .
```

You'll also see a class **`PortalInfo`** that you can ignore (it is only used within the **`StringUtil.createFromStringArray`** method) and a class **`MoveRecord`**, that you cannot ignore, but we will get back to it later.

Overview of the `PearlUtil` class

The class **`PearlUtil`** is a "utility" class, meaning that it is stateless (has no instance variables). It contains methods to implement the key algorithms for moving the player. It is almost the same as **`mini2.PearlUtil`**, with the following modifications:

- The methods are non-static.[1]
- The **`movePlayer`** method includes a **`Direction`** parameter. This is used to determine, when the player encounters spikes, whether the spikes are actually pointed at the player.
- The **`movePlayer`** and **`moveBlocks`** methods include a **`MoveRecord[]`** parameter.

When the player is moved, it may travel in one of four directions, may wrap around to the opposite side of the grid, and may "jump" to an arbitrary position in the grid via a portal. It may also cause movable blocks to shift and possibly to merge and disappear. Rather than implement the algorithms for movement four different times, continually worrying about different directions, portals and wrapping of indices, we use the following strategy:

1. When a move is initiated in a given direction, copy the affected cell states into a 1D array starting at the current player position and working cell by cell in the given direction
2. Pass the array to methods in **`PearlUtil`** that do the actual movement of states in the 1D array, now shifting them left to right only within this 1D array
3. Copy the modified array back into the 2D grid, again starting at the original player location and working in the original direction of the move

The algorithm for step 2, which is implemented in **`PearlUtil`**, is split into two parts, **`moveBlocks`** and **`movePlayer`**. The second argument to **`movePlayer`** (and likewise to **`moveBlocks`** and **`collectPearls`**) is an array of **`MoveRecord`** objects, which can assume to be **`null`** for now. The third argument to **`movePlayer`** is a *direction*, indicating which direction the player is actually moving in the 2D grid. This is used when the boundary of the cell sequence is spikes, in which case you have to check the direction of the spikes vs the direction of the player to see whether the player is impaled on the spikes. (See the method **`State.spikesAreDeadly`**.).

Overview of the Pearls class

Basic game play: the move() method

The basic play of the game takes place by calling the method

```
public MoveRecord[] move(Direction dir)
```

which moves the player in the indicated direction. As described in the discussion of **`PearlUtil`**, this is broken down into three fundamental steps, the first of which is to copy the affected cell states into a 1D array to be modified by the **`PearlUtil`** methods.

The method `getStateSequence`

[1] Since the class is stateless, these methods could have been defined to be static. However, we have specified them as non-static methods so that in testing your Pearls class we can use our own working instance of PearlUtil.

The creation of this 1D array of cells is done by the method **`getStateSequence`**. It starts at the player's current position, and follows the cells in the given direction until a boundary is reached. (The resulting sequence of states will satisfy the **`PearlUtil`** method **`isValidForMoveBlocks`**). As always, let us start with a concrete test case (you can find this code in the default package of the sample code.)

```
String[] test = {
  "##.###",
  "##$..#",
  "#.#..#",
  "#.@@.#",
  "#....#",
  "##.###"
};

Pearls game = new Pearls(test, new PearlUtil());
StringUtil.printGrid(game);
State[] states = game.getStateSequence(Direction.UP);
System.out.println();
StringUtil.printStateArray(states, 0);
```

which results in the output:

```
# # . # # #
# # $ . . #
# . # . . #
# . @ @ . #
# . . . . #
# # . # # #

$ . . . @ #
```

You can see that following direction UP from the player's original position at (*row, column*) = (1, 2), we go through cell (0, 2), then wrap around to (5, 2), (4, 2), (3, 2) and end with the wall cell at (2, 2). These are the states that you want to put into the returned array, in that order.

If there is a portal, it gets even more interesting.

```
String[] testPortals = {
  "######",
  "#Ooo$#",
  "# @@ #",
  "#    #",
  "#@O ##",
  "#  ###",
  "######"
};
```

```
Pearls game = new Pearls(testPortals,
   new PearlUtil());
StringUtil.printGrid(game);
State[] states = game.getStateSequence(Direction.LEFT);
System.out.println();
StringUtil.printStateArray(states, 0);
```

This produces output,

```
# # # # # #
# O o o $ #
# . @ @ . #
# . . . . #
# @ O . # #
# . . # # #
# # # # # #

$ o o O O @ #
```

Moving left from the player position at (1, 4) we go through (1, 3), (1, 2), (1, 1). Since (1, 1) is a portal, we jump to its companion portal at (4, 2) and keep moving left to (4, 1) and (4, 0). Recall that the **`Cell`** type includes two methods **`getRowOffset`** and **`getColumnOffset`**, which is where we get the information about the location of the companion portal. For example, if we run the code,

```
Cell c = game.getCell(1, 1);
System.out.println(c.getRowOffset() + ", " + c.getColumnOffset());
c = game.getCell(4, 2);
System.out.println(c.getRowOffset() + ", " + c.getColumnOffset());
```

the output is

```
3, 1
-3, -1
```

You can see that we can always get from one portal to the companion by adding the row and column offsets, so that is what we do when we get to the first portal. You can also see in this example that when you get to the *second* portal of a pair, you *don't* want to add the offsets, which would just take you back to the previous portal! That is something you'll have to keep track of in your code.

Logically, what you are trying to do would look like this in pseudocode:

start with (row, column) at player location
while (row, column) is not a boundary state
    add state at (row, column) to my list of states
    update row and column to be the next cell in the given direction (possibly a portal jump)

The specification breaks out the last line - updating the row and column - into a pair of helper methods:

```
public int getNextColumn(int row, int col, Direction dir, boolean doPortalJump)
public int getNextRow(int row, int col, Direction dir, boolean doPortalJump)
```

To account for the wrapping behavior, you can just use modular arithmetic. For example, when incrementing a row, the expression `(row + 1) % getRows()` will always correctly wrap around to 0 when `row + 1` goes over the height of the grid. Unfortunately you can't do exactly the same thing when *decrementing* a row or column index. In this example, when row is 0 and you decrement it, you want to wrap around to index 6 (since there are 7 rows), but the expression `(row - 1) % getRows()` just gives you -1. (In Java the modulus of a negative number is always negative.). The easiest way to work around this is just to always add `getRows()` *before* taking the modulus. Try it.

There is one potential gotcha with the helper methods. If you have two variables `row` and `col` representing your current row and column, you can't just write

```
row = getNextRow(row, col, dir, false);
col = getNextColumn(row, col, dir, false);
```

to update them, because in the second line you'd be using the *updated* `row` value, not the original. You would have to write something like,

```
int tempRow = getNextRow(row, col, dir, false);
int tempCol = getNextColumn(originalRow, col, dir, false);
row = tempRow;
col = tempCol;
```

The method `setStateSequence`

Step three of the basic algorithm is to take the cells that were modified by `PearlUtil.moveBlocks` and `PearlUtil.movePlayer` and put them back into the grid in the correct locations. That is the responsibility of `setStateSequence`. The fundamental loop for iterating through the cells is identical to the one you developed for `getStateSequence`. However, the method has a third parameter indicating which index in the sequence represents the new player position. (As for `getStateSequence`, the starting location is always the player's *current* position.)

Here's a simple test case:

```
Pearls game = new Pearls(testPortals,
    new PearlUtil());
State[] test = StringUtil.createFromString(".xxOO.#");
game.setStateSequence(test, Direction.LEFT, 5);
System.out.println();
StringUtil.printGrid(game);
```

which results in

```
# # # # # #
# O x x . #
# . @ @ . #
# . . . . #
# $ O . # #
# . . # # #
# # # # # #
```

You can assume that the given cell sequence is *structurally consistent* with the grid, in particular, that it does not attempt to move states that are the portals or walls.

Putting it together

Overall the **`move`** method of **`Pearls`** does the following things.

> *gets the state sequence array*
> *[creates a parallel array of MoveRecords]*
> *calls moveBlocks [passing in the MoveRecords array]*
> *calls movePlayer [passing in the same MoveRecords array]*
> *sets the updated state sequence and player position*
> *updates the score and move count as needed*
> *[returns the MoveRecords array]*

The **`MoveRecord`**s are discussed below. Note before you figure out the **`MoveRecord`**s, you should be able to get everything else working: just pass a null array to **`moveBlocks`** and **`movePlayer`**, and just have your **`move()`** method return null. The only thing in the overall system that is dependent on the **`MoveRecord`**s is the ability of a GUI to animate the movement of the cells.

The `MoveRecord` class

Everything related to correctly updating the game state can be implemented and tested without worrying about the **`MoveRecord`**s, and you can play the game using the console UI or even the

GUI. By default, either of the UIs just re-displays your game grid after a move is completed. The purpose of the **`MoveRecord`**s is to provide information needed for a user interface to *animate* the motion of the movable blocks and the player. In order for a UI to animate the changes, your code must provide information about *how* the grid has changed. Remember that each move is based on taking a state sequence from the grid and modifying it by changing the player position, moving movable cells, and possibly causing pearls or movable cells to disappear. The **`MoveRecord`** represents a record of what happened to each of the states.

A **`MoveRecord`** object just encapsulates:

- a **`Cell`**
- an integer **`index`** - this is the original position of the state within the state sequence
- an integer **`movedTo`** - this is the new position of the state within the state sequence
- a boolean **`disappeared`** - this flag is set if the state is removed
- a boolean **`closed`** - this flag is set if the state was an open gate that was closed by the move

Initially we start by creating an array of **`MoveRecord`** objects that is "parallel" to the array of Cell objects obtained from **`getStateSequence`**; that is, an array the same length such that its *i*th element is a **`MoveRecord`** containing the state of the *i*th Cell. Then, within the **`moveBlocks`** or **`movePlayer`** method, each time a cell is moved, the corresponding **`MoveRecord`**'s **`movedTo`** attribute is updated, and if a cell is deleted, the corresponding **`MoveRecord`**'s **`disappeared`** attribute is set. Finally, the **`movedTo`** attribute of the first **`MoveRecord`** in the array is set to the new index of the player. Let us try writing a test case:

```
PearlUtil util = new PearlUtil();
String test = "..@.o..@.#";
State[] states = StringUtil.createFromString(test);
MoveRecord[] records = new MoveRecord[states.length];
for (int i = 0; i < states.length; ++i)
{
  records[i] = new MoveRecord(states[i], i);
}
StringUtil.printStateArray(states, 0);
System.out.println();
util.movePlayer(states, records, Direction.DOWN);
StringUtil.printStateArray(states, 8);
System.out.println();
for (int i = 0; i < records.length; ++i)
{
  System.out.println(i + " " + records[i].toString());
}
```

This should produce the output:

```
$ . @ . o . . @ . #
. . . . x . . . $ #

0 [. moved to 8]
1 [.]
2 [@ disappeared]
3 [.]
4 [o closed]
5 [.]
6 [.]
7 [@ disappeared]
8 [.]
9 [#]
```

When there are movable blocks in the sequence, the **`MoveRecord`** also indicates where each one was moved:

```
PearlUtil util = new PearlUtil();
String test = "..@.o+@+-+.o";
State[] states = StringUtil.createFromString(test);
MoveRecord[] records = new MoveRecord[states.length];
for (int i = 0; i < states.length; ++i)
{
  records[i] = new MoveRecord(states[i], i);
}
StringUtil.printStateArray(states, 0);
System.out.println();
util.moveBlocks(states, records);
StringUtil.printStateArray(states, 8);
System.out.println();
for (int i = 0; i < records.length; ++i)
{
  System.out.println(i + " " + records[i].toString());
}
```

which would produce this result:

```
$ . @ . o + @ + - + . o
. . @ . o . . . $ + + o

0 [.]
1 [.]
2 [@]
3 [.]
4 [o]
5 [+ moved to 9]
6 [@ disappeared]
7 [+ moved to 10]
8 [- moved to 10 disappeared]
9 [+ moved to 10 disappeared]
10 [.]
11 [o]
```

The text-based UI

The **`ui`** package includes the class **`ConsoleUI`**, a text-based user interface for the game. It has a **`main`** method and you can run it after you get the needed game features implemented. The code is not complex and you should be able to read it without any trouble. It is provided for you to illustrate how the classes you are implementing might be used to create a complete application. Although this user interface is very clunky, it has the advantage that it is easy to read and understand how it is calling the methods of your code. Edit the main method to try out different grids; there are a few examples provided as **`static final`** variables in at the top of the file.

The GUI

There is also a graphical UI in the **`ui`** package. (*Note: It will not compile with the skeleton code until you provide stubs for some of the missing methods of Pearls.*) 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. It is provided just for fun.

Please, 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 GUI code itself is ten times more complex than the game itself and likely has bugs of its own that we are not going to fix. Use simple test cases that you can write yourself, like the examples found in **`SimpleTest.java`**. ***In particular, when we grade your work we are NOT going to run the UI, we are going to verify that each method works according to its specification.***

The controls for the graphical UI are the four arrow keys. Pressing an arrow key just invokes the game's `move()` method in the corresponding direction.

The main method is in `ui.GUIMain`. Once you get a skeleton of your code with all methods in place, you can try running it, and you'll see the initial window, but until you start implementing the required classes you'll just get errors. All that the main class does is to initialize the components and start up the UI machinery.

There are some sample grids defined in `ConsoleUI` that you can try. Edit the main method to use a different one.

If you are curious to explore how the GUI works, you are certainly welcome to do so. In particular it is sometimes helpful to look at how the GUI is calling the methods of the classes you are writing. The class `GamePanel` contains most of the GUI code and defines the "main" panel, and there is also a much simpler class `ScorePanel` that contains the display of the score and move count. The 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. If you want to see what's going on, you might start by looking at `MyKeyListener`. (This is an "inner class" of `GamePanel`, a concept we have not seen yet, but it means it can access the `GamePanel`'s instance variables.)

If you are interested in learning more about GUI development with Swing, 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 tests for your code as you develop it.

Do not rely on the UI code for testing! *When we grade your work we are NOT going to run the UI, we are going to verify 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.

> See the document "SpecChecker HOWTO", which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links" if don't remember how to import and run a SpecChecker.

Importing the sample code

The sample code includes a partial skeleton of the two classes you are writing. It is distributed as a complete Eclipse project that you can import. **There will be compile errors in `ui.GamePanel.java` until you have added stubs for all specified methods**. *Moreover, neither of the UIs will not run correctly until you have implemented the basic functionality of* **`Pearls`**.

1. Download the zip file. 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.

If for some reason you have problems with this process, you can construct the project manually as follows:

1. Unzip the zip file containing the sample code.
2. In Windows Explorer or Finder, browse to the src directory of the zip file contents
3. Create a new empty project in Eclipse
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.

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 functional tests that we run and partly on the grader's assessment of the quality of your code. 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 lots of unnecessary instance variables
    - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.
- Avoid code duplication. For example, the algorithm for shifting the player should be implemented ONLY in **`PearlUtil`** method **`movePlayer()`**. The actual method **`move()`** must not duplicate that logic.
- Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. Use internal comments where appropriate to explain how your code works. (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 a comment explaining how it works.)

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:

- 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.

- Try not to embed numeric literals in your code. Use the defined constants wherever appropriate.
- 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, and **for this assignment you are welcome to post and discuss test code**. (In the Piazza editor, use the button labeled "code" to have the editor keep your code formatting. You can also use "pre" for short code snippets.)

If you have a question that absolutely cannot be asked without showing part of your source code, change the visibility of the post to "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.)

Getting started

At this point we expect that you know how to study the documentation for a class, write test cases, and develop your code incrementally to meet the specification. *In particular, for this assignment we are not providing a specchecker that will perform any functional tests of your code.* It is up to you to test your own code (though you are welcome to share test cases on Piazza). **There are several examples of simple tests in the previous sections to get you started.**

Here are some basic observations from reading the spec:

- **`PearlUtil`** is fully implemented except for the code you will add to update the **`MoveRecord`** objects.
- You can implement and test all of **`Pearls`**, except for the **`MoveRecord`**s, using **`PearlUtil`** as it is. When you initially call **`moveBlocks`** and **`movePlayer`** in the move method. just pass null for the **`MoveRecord[]`** parameter.
- There are many methods in **`Pearls`** that are very easy accessors, such as **`getMoves`**, **`getScore`**, **`getCurrentRow`**, and **`getCurrentColumn`**. You might as well define whatever instance variables you need and implement these. Remember to update the constructor as needed whenever you define an instance variable.
- The method **`countPearls`** is very straightforward, and after that, so are **`isOver`** and **`won`**.
- You might then develop and test the **`getNextRow`** and **`getNextColumn`** methods, which are needed for **`getStateSequence`** and **`setStateSequence`**.

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, **`Pearls.java`** and **`PearlUtil.java`**. Please LOOK at the file you upload and make sure it is the right one!

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", 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 two files **Pearls.java** and **PearlUtil.java**. You can accomplish this by zipping up the **src** directory of your project. **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.