Com S 227
Spring 2020
Miniassignment 2
50 points

Due Date: Friday, October 16, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Oct 15)

10% penalty for submitting 1 day late (by 11:59 pm Oct 17)

No submissions accepted after October 17, 11:59 pm

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus 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 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.

Note: This is a miniassignment and the grading is automated. If you do not submit it correctly, you will receive at most half credit.

Overview

This miniassignment serves a couple of purposes. In part, you will be practicing writing loops that manipulate arrays. However, the algorithms you are writing will also be incorporated into Homework 3. The miniassignment includes several key operations that are isolated from Homework 3 so they can be developed and tested independently.

You will write all the specified methods of the class **`PearlUtil`**.

For detailed method specifications, see the online javadoc.

Each method operates on an array of `State` values. `State` is an *enumerated type*, or `enum`, consisting of the following constants:

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

(For the purposes of this miniassignment, we will ignore the four values `SPIKES_LEFT`, `SPIKES_RIGHT`, `SPIKES_DOWN`, and `SPIKES_UP`.) We have not used enumerated types before, but they are easy to work with. Each value is simply a named constant that you can use. New ones can't be instantiated; you would simply refer to the `EMPTY` value, for example, as `State.EMPTY`. They can be compared with `==` and `!=`. A variable of type `State` can be `null`. If you want to avoid having to type "`State.`" all the time, add the line

```
import static mini2.State.*;
```

to the top of your class.

The `State` class is provided for you; do not modify it.

What going on with `movePlayer` and `moveBlocks`??

Although you can implement the methods completely just from the specification given in the javadoc, it is probably helpful to have some intuition about what's going on.

Here is what the states mean. Imagine a player moving across a sequence of cells in a grid. Each cell has a state. The player can pass through *empty cells*, *pearls*, *open gates*, or *portals*, but not the others. Passing through a pearl causes it to disappear (i.e., it is collected by the player) and passing through an open gate causes it to become a closed gate. In this context passing through a portal has no effect, so you can essentially ignore them. A *wall*, *closed gate*, or *spikes* is always a boundary through which the player can't pass. The player can't pass through *movable blocks* either (`MOVABLE_POS`, `MOVABLE_NEG`), but their behavior is more complex because they can move. More on that later.

The basic action of **`movePlayer`** is that you are given an array of states fulfilling some *validity* criteria, such as the fact that there must be just one boundary state and it is at the end. The player is always assumed to start at index 0 and move as far to the right as possible. The tasks of **`movePlayer`** are to

- update the array by removing pearls and closing gates, and
- return the new index where the player would end up

In order to talk about examples, and to write tests, it's useful to have a character representation for the **`State`** values. We will use these, as defined in the array **`State.TEXT`**.

```
'.', //  EMPTY,
'#', //  WALL,
'@', //  PEARL,
'o', //  OPEN_GATE,
'x', //  CLOSED_GATE,
'+', //  MOVABLE_POS,
'-', //  MOVABLE_NEG,
'<', //  SPIKES_LEFT,
'>', //  SPIKES_RIGHT,
'v', //  SPIKES_DOWN,
'^', //  SPIKES_UP,
'*', //  SPIKES_ALL,
'O'  //  PORTAL;
```

Using the character representation defined above, if we start with the state sequence

```
. . @ . o @ @ #
```

then after **`movePlayer`**, the sequence should be

```
. . . . x . . #
```

and the method should return 6, which is the index where the player would end up.
(*Note: we're showing an extra space between characters in the strings shown above, for readability*). If the boundary at the end of the array is spikes (e.g. **`SPIKES_ALL`**), the behavior is a little bit different, since the player can land on the spikes (i.e. the player dies). So given

```
@ @ . . *
```

then after **`movePlayer`**, the sequence should be

```
. . . . *
```

and the method returns 4, the last index of the array, because the player landed on the spikes.

A valid sequence passed into to **`movePlayer`** may also include movable blocks, but only if they are already as far as possible to the right (i.e., so they can't move any more). The player can't pass through them. For example, given

```
@ @ . + + *
```

then after **`movePlayer`**, the sequence should look like

```
. . . + + *
```

and the method returns 2.

The last few details about **`movePlayer`** that might be needed are:

- The last state in the array might be a portal or open gate. Even though in general, the player can pass through a portal or open gate, in the context of this method the last position in the array is always treated as a boundary and the player would stop on the next-to-last state.
- If the player lands on an open gate, it does not change to a closed gate.
- The player can land on a portal.

Next we turn to the movable blocks. The basic idea is that the player will "push" the blocks to the right as far as possible. Blocks can't pass through walls, closed gates, or spikes, as it is for the player, but in addition, blocks can't go through open gates or portals. The purpose of **`moveBlocks`** is to shift all blocks as far to the right as possible. If they pass over pearls, the pearls are collected (i.e. changed to empty states). For example, given

```
. @ + . @ + @ o
```

after **`moveBlocks`**, the sequence would be

```
. @ . . . + + o
```

Notice that there are two kinds of movable blocks, labeled with a "parity" of positive or negative. If two blocks of opposite parity are pushed together, they merge together and disappear. Thus, if you had

```
. @ + - @ + @ #
```

then after **`moveBlocks`**, the sequence would be:

```
. @ . . . . + #
```

where the rightmost **`MOVABLE_POS`** has merged with the **`MOVABLE_NEG`** to its left. The algorithm works from the right side. Suppose that *end* is the next-to-last index in the array. You would search for a movable block at or to the left of *end*; if there is one, one of two things will happen:

- there is another movable block of opposite parity to *its* left, or
- there isn't.

In the first case, the two blocks are removed from the array, i.e., logically they have both moved to position *end* and have merged together and disappeared. Then you would resume searching from *end* for movable blocks to the left. In the second case, the movable block is placed at *end*, and you start searching again at *end - 1*. It's useful to have a helper method that, given a starting index, finds the index of the rightmost movable block to the left of that index. That is what you will provide in the **`findRightmostMovableBlock`** method.

String conversion methods

The **`State`** class includes some methods for converting back and forth between **`State`** values and the character representation used above. There are also methods **`toString`** for converting an array of **`State`** into a string. One of the first things you'll want to do is to implement the method **`createFromString`**, which constructs a **`State`** array from a string of characters. Then it becomes easy to write other simple test cases, such as:

```
public class SimpleTest
{
  public static void main(String[] args)
  {
    State[] s = PearlUtil.createFromString("@ . #");  // extra spaces ignored
    System.out.println(s[0]); // PEARL
    System.out.println(s[1]); // EMPTY
    System.out.println(s[2]); // WALL

    State[] states = PearlUtil.createFromString(". . @ . o @ @ #");
    int newIndex = PearlUtil.movePlayer(states);
    System.out.println(State.toString(states));
    System.out.println("Expected: . . . . x . . #");
    System.out.println(newIndex);
    System.out.println("Expected 6");
  }
}
```

Tip: it is easy to remove spaces from a string using **`replace(" ", "")`**.

Advice

- Read the documentation for the **`State`** class. It has some methods that will be useful to you.

- A normal human being is unable to write an algorithm such as **`moveBlocks`** entirely in their heads! So don't expect to sit down at the computer and just type it out. You will waste hours that way.

- Before you write any code, start out with a blank piece of paper. Work out some examples by hand so you are sure you know what you want the code to do. What steps are you following when you solve the problem on paper? Can you describe what you're doing using words such as "*while*...", "*for each*..." and "*if*..." ? Can you turn your description into pseudocode?

The SpecChecker

A SpecChecker will posted for this assignment that will perform a few dozen functional tests. However, when you are debugging, it is usually helpful if you have simple test cases of your own, as shown above.

Since no test code is being turned in, you are welcome to post your tests on Piazza for others to use and comment on.

Documentation and style

Since this is a miniassignment, the grading is automated and in most cases we will not be reading your code. Therefore, there are no specific documentation and style requirements. However, writing a brief descriptive comment for each method will help you clarify what it is you are trying to do.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`miniassignment2`**. 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 **`miniassignment2`**. *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.

What to turn in

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_mini2.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`mini2`**, which in turn contains two files, **`PearlUtil.java`** and **`State.java`**. **Always LOOK in the zip file the file to check**.

Submit the zip file to Canvas using the Miniassignment2 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 linked on the Canvas front page.

We strongly recommend that you just submit the zip file created by the specchecker. If you mess something up and we have to run your code manually, you will receive at most half the points.

> We strongly 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 **mini2**, which in turn should contain the files **`PearlUtil.java`** and **`State.java`**. You can accomplish this 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.