Project 2 - Tower of Hanoi

CS 0401 — Intermediate Programming using Java

Check the Due Date on the CourseWeb

Tower of Hanoi

(Modified from Wiki) The **Tower of Hanoi** is a mathematical game or puzzle. It consists of three poles and a number of disks of different sizes, which can slide onto any pole. The puzzle starts with the disks in a neat stack in ascending order of size on pole number 1 (left-most pole), the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another pole, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty pole.
3. No larger disk may be placed on top of a smaller disk.

With 3 disks, the puzzle can be solved in 7 moves. The minimal number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$, where n is the number of disks. The following is an example of a Tower of Hanoi puzzle:

The Goal

The final goal of this project is for you to create a Tower of Hanoi puzzle game with any number of disks so that a user can play on his/her computer. No worry, it is going to be a text mode puzzle. Not a graphic one.

The main goal is for you to practice Object-Oriented Programming. We are going to look at a Tower of Hanoi puzzle as a physical object and try to write our program using it as a guideline.

For now, let's look at the Tower of Hanoi puzzle shown earlier and see what physical objects are in the picture.

- **Disks**: There are a number of disks with different sizes.
- **Poles**: There are three poles.
- **The Puzzle**: The whole puzzle (Tower of Hanoi) itself that consists of a number of disks and three poles.

So, we are going to implement our program base on the above observation. **Note** that we are going to simulate the situation where your classes may be used by other programmers. In other words, you need to protect your class in case of any unusual situations. These situations will be discussed in each class.

Part I: The `Disk` Class (10 Points)

A most important characteristic of a disk is its size. Recall that in a Tower of Hanoi puzzle, there are a number of disks but none of them have the same size. Because of this, the size should be a part of its constructor to construct a disk with a given size.

The other noticeable characteristic of a disk is that, once you construct a disk, you cannot change its size. You need to destroy it or sand it (which is the same as reconstruct it). So, the `Disk` class will be what we usually call an **immutable** class. A user cannot change the data inside the object of type `Disk` once it has been constructed. However, users can still read data out of it.

So, from the above observations, the `Disk` class should have the following:

- **Constructors**:
  - `public Disk(int aDiskSize, char aDiskChar, char aPoleChar)`: This constructor allows a user to construct a disk with any size **greater than 0**. The `aDiskChar` and `aPoleChar` parameters will be used to generate the `String` representation of an object of type `Disk`. This will be discussed later.
  - `public Disk(int aDiskSize)`: This constructor constructs a disk with a give size and use the asterisk character (`*`) as the disk character and use the vertical bar character (`|`) as the pole character.
  - **Unusual Situations**: A disk cannot have size 0 or less. However, since your `Disk` class may be used by another programmer, you have to protect your class. There is nothing to prevent a programmer to construct a disk with a negative number or zero for the disk size as shown below:

```
Disk aDisk = new Disk(-4, '*', '|');
```

Note that a constructor cannot return a value. So, there is no way for your class to signal to a programmer that he/she construct a `Disk` object with an invalid argument. So, for this project, if a programmer construct a disk with a size 0 or less, you simply construct a disk with size 1.

- **Accessor Methods**:
  - `public int getSize()` returns the size of the disk in integer.
  - `public String toString()` returns the string presentation of a disk. The `String` representation of a disk of size n is a `String` starting with n `aDiskChar`, followed by the `aPoleChar`, and followed by another n `aDiskChar`. For example, the following code snippet

```
Disk aDisk = new Disk(5, 'x', 'M');
System.out.print(aDisk);
```

    should produce output as follows:

```
xxxxxMxxxxx
```

    For another example, the following code snippet

```
Disk aDisk = new Disk(3);
System.out.print(aDisk);
```

    should produce output as follows:

```
***|***
```

- **Instance Variables**: Obviously, you need a variable of type `int` to store the size of the disk. Having a string representation of the disk ready to be returned when a user call the `toString()` method is also a good idea. So, you are going to have at least two instance variables:

```
private int diskSize;
private String diskString;
```

  Note that the `diskString` should be constructed in the constructor. You can also have more instance variables if you want to.

A test program (`DiskTester.class`) is provided. Simply compile your `Disk.java` and make sure that `DiskTester.class` is located in the same directory as your `Disk.class`. Then simply run

```
java DiskTester
```

Part II: The `Pole` Class (20 Points)

The main characteristic of a pole is that it has an ability to hold a number of disks. A disk can be placed to a pole and can be removed from a pole. Unfortunately, a pole has its limitation. Once you construct a pole, it has finite width and height. In other words, it can only hold a finite number of disks which cannot be changed. Some disks may be too big to be placed into a pole as well. A pole can also tell you how many disks are there currently in the pole. From this observation, the `Pole` class should have the following:

- **Constructors**:
  - `public Pole(int aMaxNumberOfDisk, int aMaxDiskSize, char aPoleChar)`: constructs a pole where the maximum number of disks is `aMaxNumberOfDisk`, the maximum disk size that it can hold is `aMaxDiskSize`, and the pole character is `aPoleChar`.
  - `public Pole(int aMaxNumberOfDisk, int aMaxDiskSize)`: constructs a pole where `aMaxNumberOfDisk` is the maximum number of disks, `aMaxDiskSize` is the maximum disk size, and the pole character is the vertical bar character.
  - `public Pole(int aMaxNumberOfDisk)`: constructs a pole with where the maximum number of disks and the maximum disk size is `aMaxNumberOfDisk` and the pole character is the vertical bar character.
  - **Unusual Situations**: Obviously, if `aMaxNumberOfDisk` is less than 1, it should be set to 1. Similarly, if `aMaxDiskSize` is less than 1, it should be set to 1.
- **Accessor Methods**:
  - `public int getMaxNumberOfDisks()`: returns the maximum number of disks that this pole can hold
  - `public int getMaxDiskSize()`: returns the maximum disk size that it can hold
  - `public int getNumberOfDisks()`: returns the current number of disks that is currently holding
  - `public Disk peekTopDisk()`: returns the **reference** to the disk that is on the top of the pole **without** physically remove the disk out of the pole. Note that if the pole has no disk, this method should return the `null` reference. It is the user's responsibility to check that the return value is not equal to `null` before he/she tries to use it. Users can also check that the number of disks on this pole is not 0 (`getNumberOfDisks()`) and call the `peekTopDisk()` method.
  - `public String toString()`: returns a string representation of the pole together with disks currently on the pole. For example, consider the following code snippet (the `addDisk()` method will be discussed later):

```
Pole aPole = new Pole(5);
aPole.addDisk(new Disk(5));
aPole.addDisk(new Disk(4));
aPole.addDisk(new Disk(1));
System.out.println(aPole);
```

The output of the code snippet should be

```
     |
     |
     |
    *|*
 ****|****
*****|*****
===========
```

Note that the above string is actually the string

```
"     |     \n     |     \n     |     \n    *|*    \n ****|**** \n*****|*****\n==========="
```

without double quotations. In the above string, there are the total of 7 lines. The first line (at the top) and the last line (at the bottom) are for decoration. They represent the tip of the pole and the base of the pole. The five lines in the middle are for the maximum of five disks. Recall from the code snippet that we constructed this pole with the argument 5 which is the maximum number of disks and the maximum disk size that this pole can hold.

Note that every line actually ends with the character `\n` (newline character) except for the last line. Every line contains exactly the same number of characters. In the above case, the number of characters in each line is 11 (excluding the newline character) – Five characters (maximum disk size) on both sides of the center pole character (vertical bar character). By using the same number of characters in each line, it will simplify an implementation of a method of another class that we will discuss later.

Note that the line that contains a disk (e.g., `" ****|**** \n"`), it is a space followed by the string of the disk of size 4, and followed by another space and the newline character `\n`. The pole object does not have to create the `String` for the disk manually, simply use `toString()` method of the disk to generate the middle part (`****|****`) of a line.

- **Mutator Methods**:
  - `public boolean addDisk(Disk aDisk)`: adds a disk to the pole. **Note** that this method returns a boolean indicates whether it is successfully add a given disk. Recall that a pole has its limitation. If it is currently holding the maximum number of disks, you cannot add more disks into the pole. By calling the `addDisk()` method when the pole is full, the method should return `false`. This method also returns `false` if a user tried to add a disk with the disk size larger than the maximum disk size of the pole. It is the user responsibility to check the return value to ensure that a disk is successfully added into the pole.
  - `public Disk removeDisk()`: removes the top disk from the pole. **Note** that this method returns a **reference** to the top disk (if exists) and physically move it from the pole. If the pole is currently empty, by calling the `removeDisk()` methods, it will return the `null` reference. Thus, it is the user responsibility to check the return value that the top disk is successfully removed.
- **Instance Variables**: Since a pole needs to hold a number of disks in some order, we definitely need an array of disks. Having a variable named `numberOfDisks` which holds the number of disks currently in the pole will also help you with implementation. It can be used to determine which element of the array is the first available one or which disk is on top (think about array and its index). It can also help you determine whether the pole is full or empty. The maximum number of disks as well as the maximum disk size are very important as well. They help you determine whether a disk can be added or whether a disk is too big to be placed into a pole. Thus, you should have at least the following instance variables:

```
private Disk[] disks;
private int numberOfDisks;
```

```
private int maxNumberOfDisks;
private int maxDiskSize;
```

Note that you are welcome to have more instance variables if you think you need them.

Part III: The `TowerOfHanoi` Class (30 Points)

Again, we are going to think about an object of type `TowerOfHanoi` as the actual Tower of Hanoi puzzle shown above. A Tower of Hanoi puzzle consists of three poles and 7 disks (size 7 to size 1) neatly stack from bottom to top in that order.

For our Tower of Hanoi, we are going to make it a little bit more flexible. In other words, you do not have to always play the Tower of Hanoi with 7 disks. You can choose to play with any number of disks (greater than or equal to 1).

Given a Tower of Hanoi with a number of disks, a player can do the following:

- Move the top disk from one pole to another (or even put it back on the same pole)
- Check the disk that is currently on top of a given pole
- Get the number of disks currently on a given pole

We are also going to make this class easier to use. When there are a number of poles, most people generally refer to each pole as pole number 1, pole number 2, and pole number 3. Unlike computer, everything starts at 0. So, we are going to let users use pole, 1, 2, or 3 instead of 0, 1, or 2. Note that with a physical Tower of Hanoi object, nothing prevent a person to actually put a larger disk on top of the smaller one. We will enforce this rule when we create the actual game.

Again, of the above observations, the `TowerOfHanoi` class should have the following:

- **Constructors**:
  - `TowerOfHanoi()`: This is the default constructor that constructs a Tower of Hanoi objects with 7 disks where all disks are neatly stacked (according to the puzzle) in the pole number 1 (left-most pole). Note that the maximum number of disks of each pole should be 7 and the maximum disk size of each pole should also be 7.
  - `TowerOfHanoi(int aNumberOfDisks)`: This constructs a Tower of Hanoi objects where the number of disks is `aNumberOfDisks`. This will allow a user to play the Tower of Hanoi puzzle with smaller or larger than 7 disks. Similarly, the maximum number of disks and the maximum disk size of each pole should be `aNumberOfDisks`. **Note that the number of disks must be greater than or equal to 1. If not, just set it to 1**.
- **Accessor Methods**:
  - `public Disk peekTopDisk(int aPoleNumber)`: This method returns the reference to the top disk of a given pole. Note that if the given pole number is invalid (not 1, 2, or 3), this method should return the `null` reference. Similarly, if the given pole contains no disk, a `null` should be returned. It is the user's responsibility to check the return value.

– `public int getNumberOfDisks(int aPoleNumber)`: This method returns the number of disks currently on a given pole. Note that if the pole number if invalid (not 1, 2, or 3), −1 should be returned.

– `public String toString()`: This method returns the string representation of a Tower of Hanoi puzzle. For example, the following code snippet:

```
TowerOfHanoi toh = new TowerOfHanoi();
System.out.println(toh);
```

should give you the following output:

```
       1               2               3
       |               |               |
      *|*              |               |
     **|**             |               |
    ***|***            |               |
   ****|****           |               |
  *****|*****          |               |
 ******|******         |               |
*******|*******        |               |
===============================================
```

Note that there are numbers (1, 2, and 3) indicating pole number at the top of each pole. The number of characters in each line (excluding the newline character) is 47 (15 + 1 + 15 + 1 + 15) where 15 is 7 + 1 + 7 where 7 is the maximum disk size of each pole (constructed using the default constructor) and 1 is the pole character. The other two (+1s) in 15 + 1 + 15 + 1 + 15 is a space between two poles. Note that the base uses = instead of a space.

For this project, you just use strings returning from the `toString()` methods of each pole to generate the above string. **This is one of the main requirements of this class that the above string must be constructed from strings from poles**. Note that by simply concatenate all strings returning from the `toString()` methods of all three poles will not give you the above string. You may need to use the `split()` method (see `String` API) using `"\n"` as the argument together with array of strings to achieve the result.

Here is another example from the following code snippet:

```
TowerOfHanoi toh = new TowerOfHanoi();
toh.move(1, 2);
toh.move(1, 3);
toh.move(1, 2);
System.out.println(toh);
```

where the `move()` method will be discussed next and the following is its output

```
       1               2               3
       |               |               |
       |               |               |
       |               |               |
       |               |               |
```

```
   ****|****           |               |
  *****|*****          |               |
 ******|******      ***|***            |
*******|*******       *|*            **|**
===============================================
```

Note that according to the rule of the Tower of Hanoi puzzle, you cannot put disk size 3 on top of disk size 1 (see the pole number 2 above). However, this is just a Tower of Hanoi object, you can actually do anything you want. We will have another class that will enforce the rule of the Tower of Hanoi puzzle.

- **Mutator Methods**:
  - `public boolean move(int fromPole, int toPole)`: This method moves the top disk from `fromPole` to `toPole` where `fromPole` and `toPole` are 1, 2, or 3. Recall that we want this to be a user friendly class. So, pole number starts at 1 instead of 0. Note that this method return a `boolean` value. There are some unusual situation where you cannot move. For example, if the `fromPole` contains no disks, this method should return `false`. Or if the `toPole` is full, it should return `false` as well. Another unusual situation is when `fromPole` or `toPole` are not 1, 2, or 3. If it is successfully move a disk, simply return `true`.
  - `public void reset()`: This method sets the puzzle back to the original when it was constructed.
- **Instance Variables**: Obviously, you need an array of `Pole`s. This array should be initialized when a user construct a new object of the Tower of Hanoi. Some of you may think that you need the array of `Disk`s as well. Ideally, you do not need it. Yes, you need to construct a number of disks and put them into a pole. You can actually construct one disk and put it into the first pole (one at a time starting from the largest disk). Once you are done, all disks are in a pole (the pole number 1) which we be accessed by removing it out of a pole. Again, you add more instance variables into your `TowerOfHanoi` class if you think it will help you with the implementation.

Part IV: The `TowerOfHanoiPuzzle` Class (`main` class) (20 Points)

This will be the main class. In other words, this class will have only static method(s) including the `main()` method. This class will let a user play the Tower of Hanoi puzzle where rules of the puzzle will be enforced by this class. So, this program should perform the following:

1. Ask a user whether he/she wants to play the Tower of Hanoi puzzle or exit as shown below:

```
Welcome to Tower of Hanoi Puzzle
Take a pick:
  1) Play a Tower of Hanoi Puzzle
  2) Exit
```

If the user pick 2, simply exit the program. If the user pick 1, continue. However, if the user enter another number other than 1 and 2, simply ask again.

2. Ask a user how many disk he/she wants to play as shown below:

```
How many disks would you like to play (between 1 and 64): 4
```

   Note that if the user enter a number that is out-of-range (less than 1 or greater than 64), simply ask again.

3. Construct a Tower of Hanoi object where the maximum number of disks and the maximum disk size of each pole is the same as the number of disks the user wants to play

4. Show the tower of Hanoi on the screen with all disks are neatly stacked into the first pole. Also show the goal of the puzzle including the least number of possible moves and ask the user whether he/she is ready to play. The following is an example of the Tower of Hanoi puzzle with 4 disks:

```
    1         2         3
    |         |         |
   *|*        |         |
  **|**       |         |
 ***|***      |         |
****|****     |         |
=============================

The goal is to move all 4 disks from pole 1 to pole 3
The least number of moves for 4 disks is 15.
Are you ready to play? (y/n):
```

   If the user enter `n`, simply go back to the main menu (step 1). If the user enter something else other than `y` or `n`, ask again.

5. Once user answer y, show the current puzzle together with the number of moves that the user has been used so far (0 at the beginning) and ask the user to enter two numbers to move the top disk from one pole to another as shown below:

```
    1         2         3
    |         |         |
   *|*        |         |
  **|**       |         |
 ***|***      |         |
****|****     |         |
=============================
Number of Moves: 0
Enter <from><space><to> to move a disk:
```

   where `<from>` is a pole number (1, 2, or 3) and `<to>` is a pole number. If a user enter `0 0` (`<from>` is 0 and `<to>` is also 0), simply go back to the main menu.

   If user enter 1 3 (move the top disk from the pole number 1 to the pole number 3), your program should show the current puzzle, number of moves, and ask again as shown below:

```
Enter <from><space><to> to move a disk: 1 3
    1         2         3
    |         |         |
    |         |         |
  **|**       |         |
 ***|***      |         |
****|****     |        *|*
============================
Number of Moves: 1
Enter <from><space><to> to move a disk:
```

Do not forget that a user is not allowed to put larger disk on top of the smaller one. So, if the user enter 1 3 again, here is the result:

```
You cannot move the top disk from pole 1 to pole 3.
The top disk of pole 1 is larger than the top disk of pole 3.
    1         2         3
    |         |         |
    |         |         |
  **|**       |         |
 ***|***      |         |
****|****     |        *|*
============================
Number of Moves: 2
Enter <from><space><to> to move a disk:
```

Note that the number of moves should be increased as well even though the user made a mistake.

There is also a chance that the user enter something that is invalid. For example, the user may enter `1 4`.

6. Lastly, if a user is able to move all disks from pole 1 to pole 3, simply congratulate the user with his/her number of moves compared to the least number of moves as shown below:

```
Enter <from><space><to> to move a disk: 2 3
    1         2         3
    |         |         |
    |         |        *|*
    |         |       **|**
    |         |      ***|***
    |         |     ****|****
============================
Congratulation!!!
Number of Moves: 15
The least number of moves for 4 disks is 15.

Welcome to Tower of Hanoi Puzzle
```

```
Take a pick:
  1) Play a Tower of Hanoi Puzzle
  2) Exit
```

Note that the program should go back to the beginning. An output of a run is also posted on the CourseWeb named `output_toh.txt`.

Submission

The due date of this project is stated on the CourseWeb. Late submissions will not be accepted. For this project, you have to submit the following files:

- `Disk.java`
- `Pole.java`
- `TowerOfHanoi.java`
- `TowerOfHanoiPuzzle.java`

Zip them in to one file named `project2.zip` and submit the file `project2.zip` via CourseWeb.