

07.01 Project 3: Tower of Hanoi

Due Oct 17 by 11:59pm **Points** 100 **Submitting** an external tool **Available** after Sep 28 at 12am

- It is an Honor Code violation to resubmit prior work from an earlier semester in this course.
- It is an Honor Code violation to submit work authored by someone else as your own.

You must include the following Virginia Tech Honor Code pledge at the very top of each Java file you write (separate from the class Javadoc comment). Include your name and PID as shown, to serve as your signature:

```
// Virginia Tech Honor Code Pledge:
```

```
//f
```

```
// As a Hokie, I will conduct myself with honor and integrity at all times.
```

```
// I will not lie, cheat, or steal, nor will I accept the actions of those who do.
```

```
// -- Your name (pid)
```

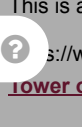
Project 3 Tower of Hanoi

The project will be graded on how well it executes. Please follow the individual UML class diagram when implementing each class of the project.

Project 3 Introduction

Refer to your course materials, the Recursion Module, and the following example videos to familiarize yourself with the famous *Tower of Hanoi* puzzle.

This is a funny but informative video of a young person solving the puzzle blindfolded. How does he know which disk to move when?? (Hint: *recursion!*)

 <https://www.youtube.com/watch?v=gctRbzepm6c&spfreload=10>

Tower of Hanoi  <https://www.youtube.com/watch?v=gctRbzepm6c&spfreload=10>



<https://www.youtube.com/watch?v=gctRbzepm6c&spfreload=10>

See below a more detailed video that includes a description in python. (You may want to skip to the 8min0sec mark. In the first eight minutes, they explain the basics of the puzzle, which you already know, and talk about a very cool way to intuit the solution using binary counting. By all means watch the whole video, especially if you're into math!)

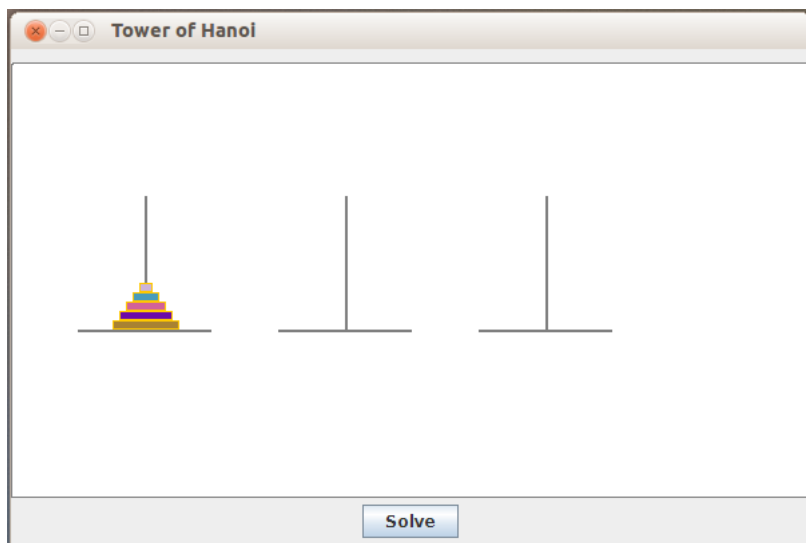
<https://youtu.be/2SUvWfNJSsM?t=480>

Binary, Hanoi and Sierpinski, part 1  <https://youtu.be/2SUvWfNJSsM?t=480>

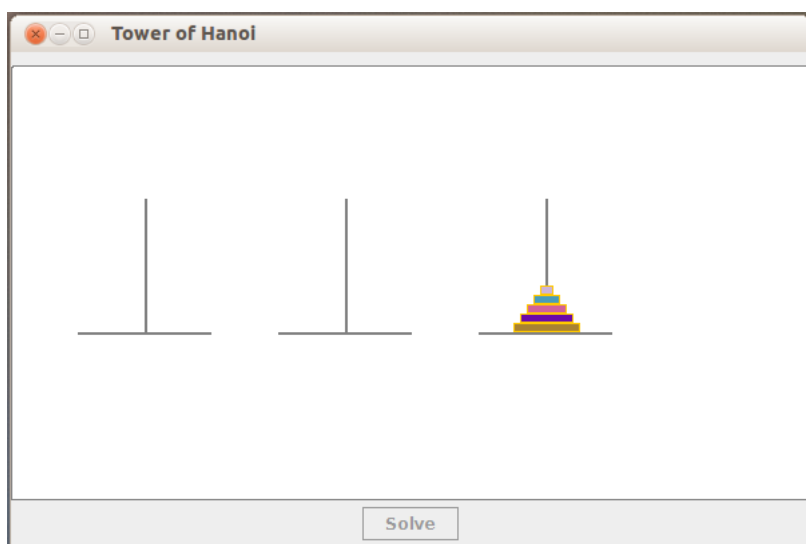


<https://youtu.be/2SUvWfNJSsM?t=480>

In this project, your goal is to implement your own [LinkedStack](#) and solve the classic Tower of Hanoi problem using recursion. Using the algorithm as described in your course materials (within the Recursion Module [7.3.12. Interactive: Tower of Hanoi](#) <https://canvas.vt.edu/courses/176115/modules/399162>), you will solve a puzzle, step by step, and display your moves in a [Window](#). You'll start the puzzle with the program display looking something like this...



and end it looking something similar to:



design of this project separates the front-end and the back-end. The back-end extends `Observable` and the front-end extends `Observer`. This design allows for separation of responsibility for the front and back ends which increases cohesion and reduces coupling. When there is a change in the observable back-end, the front-end observer is notified and can update the display accordingly.

Prerequisites

Skills from project 1 such as unit testing, webCAT submission, use of CS2GraphWindowLib and CS2DataStructuresLib

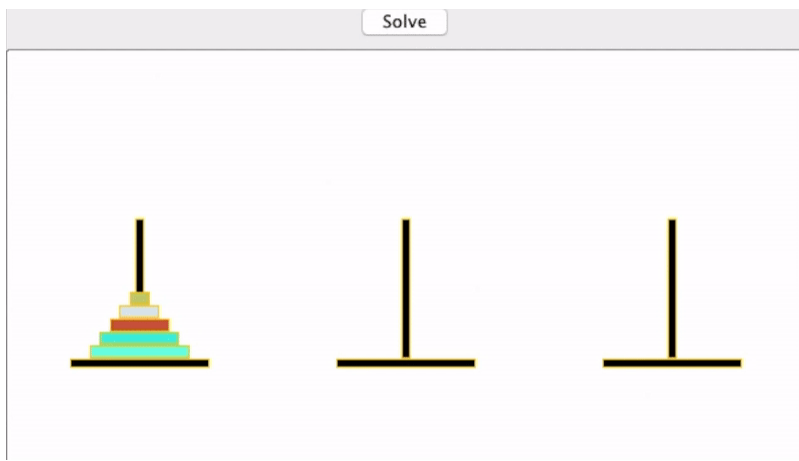
Using a data structure

Creating and manipulating linked chains

Understanding of stack

Concepts behind building a data structure with a linked chain

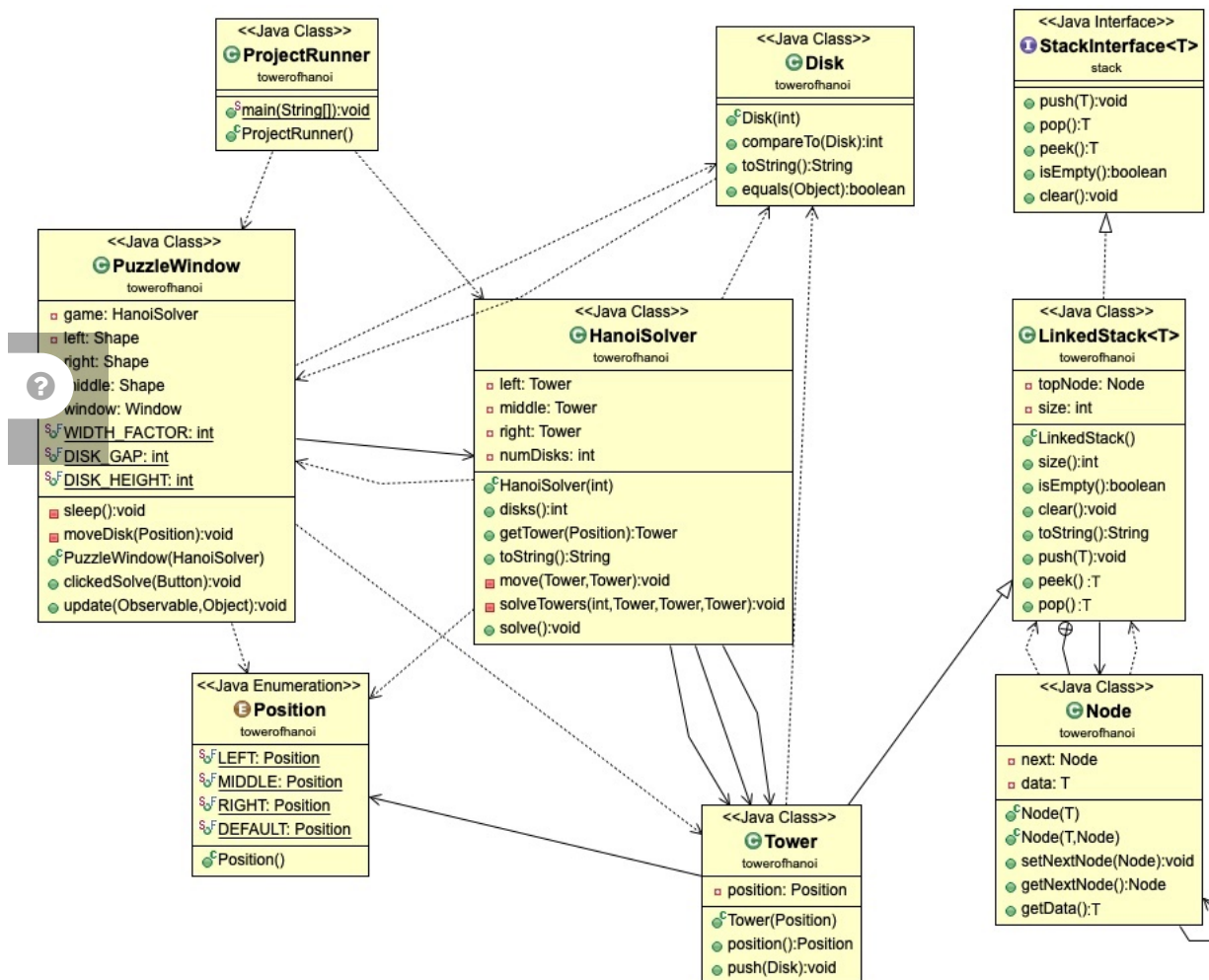
Expected Outcome



If extenuating circumstances arise for you just before a deadline, we will only take them into consideration for grading purposes if you have already submitted more than 50% of the project. As a general rule we recommend you complete at least 50% of a project 5 days before any deadline.

Project 3 Class Overview

UML.





Project 3 Steps Breakdown

Implementation Tips

Your implementation can follow the specific class diagram in each section below even when there's a slight variation from the larger diagram.

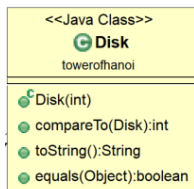
There are several classes that throw Runtime exceptions. Because these are Runtime exceptions and not checked exceptions, a `throws` clause is not necessary. However, it is recommended that conditions that cause the Runtime exceptions be listed in the class's JavaDocs with `@precondition` tags.

Create your project, Tower Of Hanoi, and set up your build path as you have in your previous projects. For your import statements, only import the classes you need; you do not want `Node` or `LinkedList` from `CS2DataStructuresLib` because you will be implementing those yourself. Notice in the UML diagram that the package name we are requiring you to use is `towerofhanoi`.

Don't forget to test as you go! We walk through the classes in the order that they rely on one another. Make sure each class is working properly before you move on to the next, or tracking down your bugs will be much harder.

Disk extends Shape implements Comparable<Disk>

The `Disk` class extends `Shape` because we are thinking of Disks as rectangles of various widths. (For a reminder about how `Shape` works, check [the API](http://courses.cs.vt.edu/~cs2114/Fall2020/CS2-GraphWindow-JavaDocs2020/cs2/package-summary.html) (<http://courses.cs.vt.edu/~cs2114/Fall2020/CS2-GraphWindow-JavaDocs2020/cs2/package-summary.html>)). Dimensions are important for determining valid moves, since only smaller disks can be placed on larger ones. `Disk` also implements the `Comparable<Disk>` interface to allow Disks to be compared to one another.



Disk(int width)

Call the `super()` constructor, which takes 4 parameters: x, y, width and height. Use coordinates (0,0) for now. The width comes from the parameter and the height should eventually come from `PuzzleWindow's DISK_HEIGHT` which can be referenced statically once that class is written (in the meantime hardcode it or create a local constant). We'll move these disks after they're built. After calling `super`, set the disk's background color to a random new `Color`. To randomly generate a color, we will use the `TestableRandom` class to generate random numbers. Import `student.TestableRandom` to use this class. Then we will call the `Color` constructor with three random integers ranging from 0 to 255 (You will need to: `import java.awt.Color;` to access `Color` class.)).

compareTo(Disk otherDisk)

For use in determining relative size of disks, our Disks are comparable to other disks. If `otherDisk` is null, be sure to throw an `IllegalArgumentException`. Otherwise, compare widths and return a negative number if this `Disk` is smaller than the disk parameter, a positive number for the opposite, and a zero if their widths are equal.

toString()

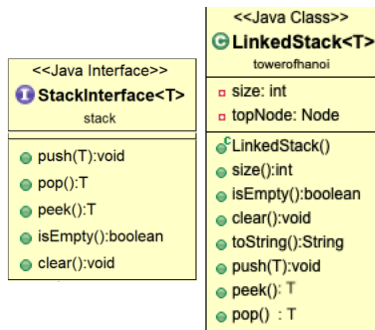
Return the width of this `Disk` as a string. Use its `getWidth()` method. For example, calling `toString()` on a disk of width 10, `disk1.toString()`, should return `"10"`.

equals(Object obj)

Two disks are equal if they have the same width. See your course material and notes for more information on how to implement an equals method.

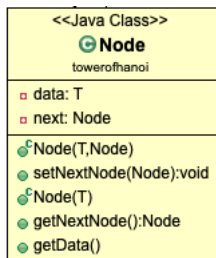
LinkedList implements StackInterface

Write a `LinkedList<T>` class that implements the given `StackInterface<T>`. You should add the `StackInterface<T>` when creating `LinkedList<T>` class (choosing the one from `stack` package, NOT from `bin.stack` package). You can view details about `StackInterface` in [the API](http://courses.cs.vt.edu/~cs2114/Fall2020/CS2-DataStructures-JavaDocs2020/overview-summary.html) (<http://courses.cs.vt.edu/~cs2114/Fall2020/CS2-DataStructures-JavaDocs2020/overview-summary.html>). Implement this stack with linked nodes. Include a size field and the `size()` method for keeping track of how many objects are in the `LinkedList`. The front-end of our project will need this information from the back-end. Be sure your project is using your `LinkedList` implementation and not the one from `CS2DataStructuresLib`.



private Node class

In the `LinkedStack` class, implement your own inner private `Node` class at the bottom of the class. These Nodes should be singly linked. To make your logic easier, we recommend making a constructor which sets both its `data` and its `nextNode` fields immediately upon creation. We leave the implementation largely up to you, since you have used a Node class before (remember project 2, and your course materials). Be sure your project is using your Node implementation and not the one from CS2DataStructuresLib.



Suggested constructors for Node that correspond with this UML diagram and simplify testing tip:

```

public Node(T entry, Node node)
{
    this(entry);
    this.setNextNode(node);
}
  
```

and

```

public Node(T data)
{
    this.data = data;
}
  
```



`LinkedStack()`

One `Node` field, named `topNode`, should be null by default. Since the stack is empty, there isn't a head node yet.

size() && isEmpty() && clear() && toString()

Implement these yourself. Make `toString()` look the same as if you were printing out an array with the same contents. For example, if `stack1` is a `LinkedStack` with 3 items in it, `stack1.toString()` should output `"[lastPush, secondPush, firstPush]"`, and if `stack1` is empty, `"[]"`. Notice the order of the output is **from top to bottom**. It is generally preferable to use a [StringBuilder](https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html) (<https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html>) for string concatenation in Java. It is much faster and consumes less memory.

The example above `[lastPush, secondPush, firstPush]` depicts both a **comma** and a **space** after each entry except for the last.

Keep this in mind when implementing and testing your `toString`. This is generally the approach we use throughout the course. For a given data structure, the `toString()` returns a String representation of the data structure, where the elements are enclosed in square brackets `[]` and are separated by a comma followed by a space.

push(T anEntry)

Push will "push" a new entry on the top of the stack. Place `anEntry` in a new `Node`, and set its `nextNode` to be your head field. This puts `anEntry` in front of your `topNode`. Finally, update `topNode` to be the new Node, and increment your size.

peek()

Peek exists to show what's on the top of the stack, without modifying the stack in any way. If the stack is empty, this method throws an `EmptyStackException`. Otherwise, return your `topNode`'s data. You'll need to import `java.util.EmptyStackException`.

pop()

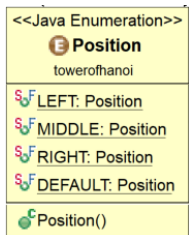
Pop will throw an `EmptyStackException` if called on an empty stack. Use `java.util.EmptyStackException`. Otherwise, your job here is to take away the `Node` from the top of the stack, and return its data. Store your `topNode` data field in a local variable. Set the `topNode` to be the `topNode`'s next node, effectively losing the original top `Node`. Luckily, you already saved it as a local variable. Decrement your size, and return the original top `Node`'s data.

enumerator Position

There are always three poles in the Towers of Hanoi puzzle, and so each `Tower` is associated with one of three positions: `LEFT`, `MIDDLE`, or `RIGHT`. Make a new Enum much like you would make a new Class, go to File > New > Enum. Name it `Position`, and click Finish. Inside our `Position` enum file, you'll see an empty code block. Inside, separated by commas, name your positions. They are...

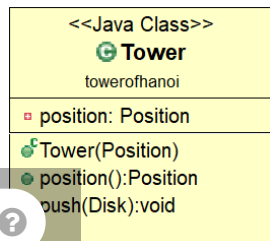
```
LEFT, MIDDLE, RIGHT, DEFAULT;
```

... and will be used to determine the `Tower` position. The constructor in this Enum is not needed.



Tower extends LinkedStack<Disk>

The `Towers` on which we store `Disks` function as stacks. We extend the `LinkedStack<Disk>` we just implemented, since Towers offer a unique extension to a normal stack - they only allow smaller disks to be placed on top of larger ones.



Tower(Position position)

Call the `super()` constructor to create our stack and then store this `Tower`'s position in a field.

position()

Return your `Tower`'s position here.

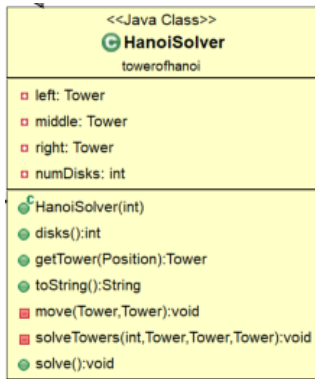
push(Disk disk)

This push method must override the `LinkedStack`'s push. Make sure you add an `@Override` tag to this method.

We need to check if it is valid to push the disk provided. If this tower is already empty, or the disk on top is larger than the disk being pushed, we have a valid push. Use the disk's `compareTo()` method to determine which is smaller. If it is a valid push, we call `super.push()`, and provide the disk. If this is an invalid push, we will throw an `IllegalStateException` or if the disk passed in is null we should throw an `IllegalArgumentException`.

HanoiSolver extends Observable

Your `HanoiSolver` represents a Tower of Hanoi puzzle. These puzzles vary in regard to how many disks are used, so the constructor requires this as a parameter. There are always three towers, so you will have three private `Tower` fields named left, middle, and right. You also extend `Observable` (by `import java.util.Observable;`), so that the `PuzzleWindow` may observe `HanoiSolver` to update the display that animates the Disks. Because `Observable` is deprecated you will notice it is shown with strikethrough in eclipse. You can dismiss this as we use this straightforward interface for introductory experience.



HanoiSolver(int numDisks)

Every Tower of Hanoi puzzle requires the number of disks to be specified. Store *numDisks* in a field, and initialize your three *Tower* fields to be new Towers, with the corresponding *Position.LEFT*, *Position.MIDDLE*, or *Position.RIGHT* provided as parameters. Leave them empty for now. The front-end will fill them in later.

disks()

Return the number of disks, *numDisks*.

getTower(Position pos)

Depending on the position requested, return either *left*, *middle*, or *right*. The enumerated type makes it straightforward to use a switch statement here.

Note that there is still the fourth enum type *default*. This is for testing purposes so that you have a way to test the *default* case in your switch statement. In the default case, you should return the middle tower.

toString()

Return *left*, *middle*, and *right*, as strings (use *toString()*), appended to each other. For example: if the left, middle, and right tower each have a single disk with width of 10, 20, and 30 respectively, the output of *toString()* is "[10][20][30]". To test *toString()* you will need to instantiate a HanoiSolver object and push disks onto its towers.

move(Tower source, Tower destination)

This method executes the specified move. Pop the *Disk* from the "source" *Tower*, and push it onto the "destination" *Tower*. Any error checking and handling will be done by the *Tower* class, such as *IllegalStateException* or *NullPointerException*. Now we need to then indicate that something has changed about this class with a call to *setChanged()*, then tell any observers that it's time to update with a call to *notifyObservers(destination.position())*. The destination tower's position is provided, to communicate with the front-end as to which *Tower* needs to be updated.

solveTowers(int currentDisks, Tower startPole, Tower tempPole, Tower endPole)

Implement your recursive *solveTowers()* method with help from your course materials. Start with the base case where there is only one disk left to move. Otherwise, you recursively solve smaller sub-problems by calling *solveTowers()* with slightly different parameters, invoking the *move()* method when it is necessary for a disk to be moved.

solve()

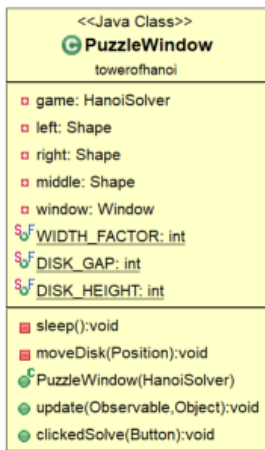
Your *solve()* method will be public. It makes the initial call to the recursive *solveTowers()* method. It provides the *solveTowers()* method the correct parameters, with *left* being the startPole, *middle* being the tempPole, and *right* being the endPole.

PuzzleWindow implements Observer

Part of this class has been written for you. Please download and drag into your project, the skeleton provided [here](https://canvas.vt.edu/courses/176115/files/29950038?wrap=1)

(<https://canvas.vt.edu/courses/176115/files/29950038?wrap=1>). [↓](https://canvas.vt.edu/courses/176115/files/29950038/download?download_frd=1) (https://canvas.vt.edu/courses/176115/files/29950038/download?download_frd=1) :

Our *PuzzleWindow* creates the *Window* in which we view the puzzle, and observes *HanoiSolver* given to it by the main method present in ProjectRunner (refer the complete UML diagram given in the beginning) for updates on when to animate the *Window*. Because this class is an *Observer* (by `import java.util.Observer;`), it requires the *update()* method. The front-end class, given (mostly) to you, has a *HanoiSolver* field named *game*, and three *Shape* fields indicating the *left*, *middle* and *right* towers on the front-end. Because *Observer* is deprecated you will notice it is show with strikethrough in eclipse. You can dismiss this as we use this straightforward interface for introductory experience.



Understanding the provided methods

The majority of the ShapeWindow class has been given to you. The `sleep()` method provides a means to pause between Disk movements. Without a pause, we wouldn't be able to see the algorithm in action. The `clickedSolve(Button button)` method supports your Solve button. The new `Thread` is needed for `clickedSolve()` so that when it calls your game's solve method, the display is updated when the back-end changes. Try to understand how they work, and read a little about Threads online.

moveDisk(Position position)

This method updates the front-end, after the back-end has been changed. We only need the position parameter so we can peek at the `Disk` that was just moved to get needed information for the display. The back-end already moved the disks between the Towers.

Make a local `Disk` variable named `currentDisk` and a local `Shape` variable `currentPole` for storing the `Disk` and pole associated with the move. Use `position` as a parameter to `getTower()` to get the current disk.

Hint, to invoke `getTower()` from within `moveDisk(Position position)` please recall the following:

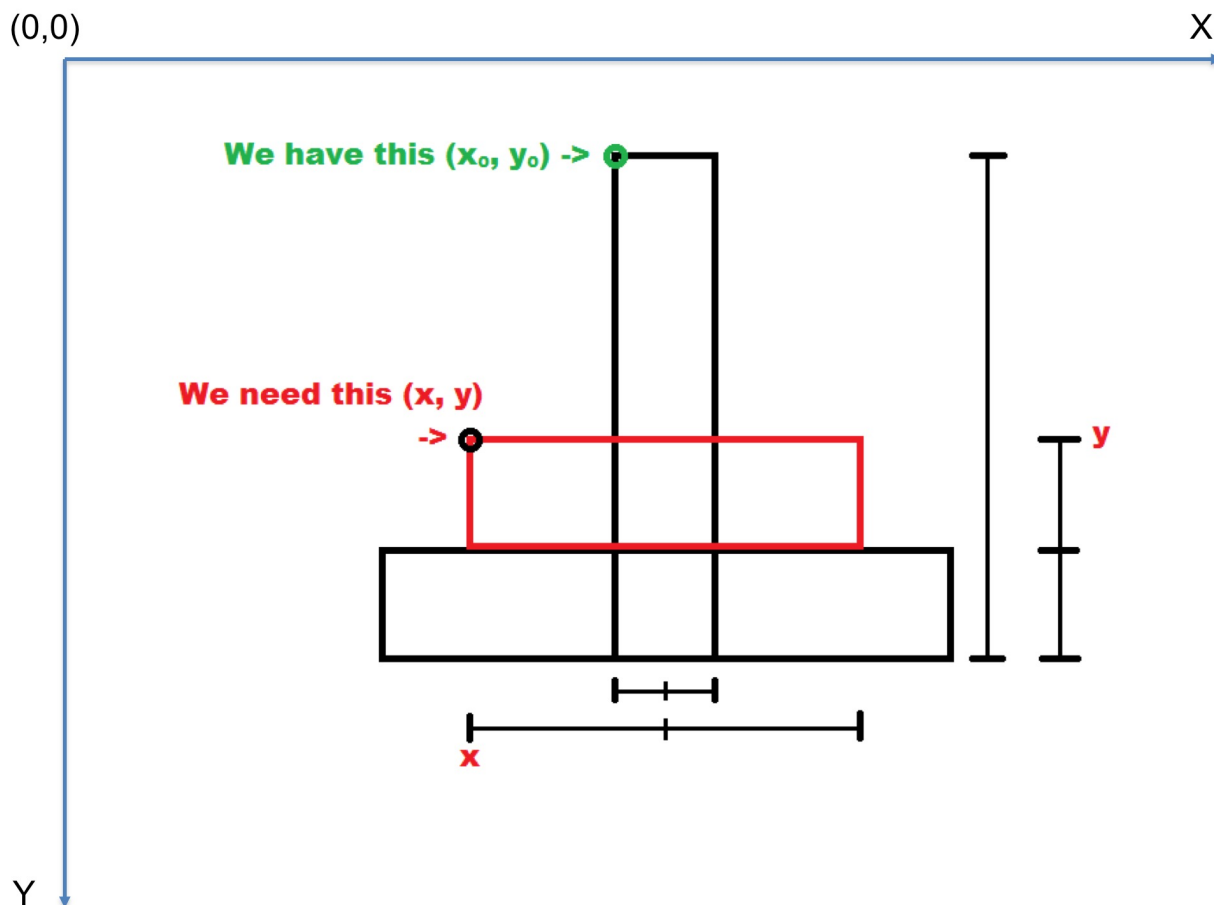
- 1) The `getTower()` method was implemented within `HanoiSolver`.
- 2) `PuzzleWindow` should include a field of type `HanoiSolver` (see UML for further detail).

Then, determine which `Shape` field corresponds to this position. If the `Position` is `LEFT`, set your local variable to be `left`, and so on.

The `moveTo()` method that `Disk` inherits from `Shape` will graphically relocate the disk in the display. Use the `getX()`, `getY()`, `getWidth()`, and `getHeight()` methods for the disk and pole to determine the new X and Y coordinates to which the disk will move. You will also need the `size()` of the tower to know how many disks are underneath the current one, to adjust its Y position appropriately. If you want to add a gap between disks or a height from the base of the pole, you can use the final attributes `DISK_GAP` and `DISK_HEIGHT` respectively in your calculations of the new



y. The following diagram may help you with coordinates:



PuzzleWindow(HanoiSolver game) (Follow carefully)

Note: You do not need to write unit tests for the `PuzzleWindow` class.

While this class has already been written for you, it's very important for you to understand it as your `moveDisk` implementation uses the fields instantiated and built by it.

Store the game parameter in a field, and call game's `addObserver(this)` method. Provide "`this`" (not a string) as its parameter to indicate that this class is observing it for updates.

```
this.game = g;
game.addObserver(this);
```

Declare a new Window as a class field. Choose a good width and height for it. You might need to adjust this as you see fit or if needed. Give the window the title "Tower of Hanoi". **(Do this yourself)**

Now we're going to initialize your `Shape` fields that represent the towers. They should be very tall and narrow rectangles, somewhat reasonably spaced in the window, with the left one being the farthest left, and so on. The `Color` and everything else is up to you. Once you see them on the display, play with the parameters. They should not affect your program's performance or test compatibility.

```
//The height and Y location of each pole are the same
int poleHeight = 400;
int poleY = (window.getGraphPanelHeight() / 2) - (poleHeight / 2);
left = new Shape((200 - 15 /*width*/ / 2),
    poleY, 15, poleHeight, new Color(50, 50, 50));
middle = new Shape((window.getGraphPanelWidth() / 2) - 15 / 2,
    poleY, 15, poleHeight, new Color(50, 50, 50));
right = new Shape((window.getGraphPanelWidth() - 200) - 15 / 2,
    poleY, 15, poleHeight, new Color(50, 50, 50));
```

In a `for` loop, based on the game's number of disks, generate the disks from largest to smallest. Make a new `Disk` with its width based on the `for` loop's index. Make the disk width a multiple of some number between 5 and 15, depending on your aesthetic preference. Next, add the `Disk` to the `Window`. Push each `Disk` onto the game's left `Tower`, then call `moveDisk(Position.LEFT)` inside the loop. This will update the disk's x,y location to the correct spot on the `Window`.

```

for (int width = (game.disks() + 1) * WIDTH_FACTOR;
    width > WIDTH_FACTOR;
    width -= WIDTH_FACTOR) {
    /* TODO: create a new disk, and add it to the left tower.
       Make sure to add the disk to the window */
}

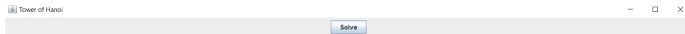
```

After the loop, add the previously created *left*, *middle*, and *right* `Shape` fields to the `Window`. We add them after the disks so that they appear underneath the disks. If you want to manipulate the ordering of a `Shape`, you can call its `bringToFront()` or `sendToBack()` methods. **(Do this yourself)**

For appearances' sake, you can also add a low and wide rectangle centered underneath each tower. This gives the appearance of the base, and looks nice. This is not required.

Now we'll deal with your button. Declare a new `Button` named "solve" that says "Solve", add it to the NORTH side of the Window, and tell it to `onClick(this, "clickedSolve")`. **(Do this yourself)**

While you might not be able to view your window quite yet, It should look something like this when done:



update(Observable o, Object arg)

Our update method is called automatically when the game's `move` method calls `notifyObservers`. In `HanoiSolver`, we passed `notifyObservers()` a `Position` as a parameter. That `Position` will be passed as our `arg` parameter here.

To start, the method checks if `arg.getClass()` equals `Position.class`. If true (meaning the argument given is indeed a `Position`), it casts the `arg` to be a `Position`. It will then call your created `moveDisk` method with the position, then call `sleep()`.

ProjectRunner

Note: You do not need unit tests for the `ProjectRunner` class.

As in previous projects, this last class is where your main method lives.

main(String[] args)

The last method! Here, we make a new `PuzzleWindow`, and pass it a new `HanoiSolver`, which we pass the number of disks to use. To determine how many disks to use, declare a local variable named `disks` that is **six** by default. If the `String` array `args` length is exactly one, set your `disks` integer to be the Integer parsed from the `args[0]` location (`disks = Integer.parseInt(args[0])`). This allows you to change this program's Run Configuration... > Arguments to any number you like, but only if there's one argument.

Optional Features:

These ideas are purely optional, for if you want to do more. Make sure that your tweaks don't break the original functionality, or you may resubmit and find your grade went down! To avoid this, get a 100% before trying these options.

Dynamically set pole and base dimensions

You might want to only make the bases and poles as large as you need to hold as many disks as the game is currently running with. The change is slightly prettier, with a little math involved.

Provide a step-by-step button

Whenever clicked, only move one step through the solve algorithm. Make sure that the solve button still works after the user has pressed the step button several times.

Speed options

To accelerate longer puzzles, adjust your sleep time based on the number of disks. Modify the provided `sleep()` method to provide this behind-the-scenes tweak.

Allow user interaction with disks

To let users solve the puzzle themselves, you can allow users to drag and drop their shapes. Provide the shapes with an onClick method, such that users can click a Disk, then click a Pole to specify a move. Make sure your step-by-step and solve buttons still work if the user performs these interactions! You also might want to make your disks bigger than in the examples to make them easier to click on.

**Submission****Web-CAT****Grading Rubric****50 pts WebCAT****50 pts Execution, Style and Documentation**