NAVIGATION

Home
- My home
  - My profile
  - Current course
    - CS 2114 Spr 2016
      - Participants
      - February 22 – February 28
        - Reading Quiz 6 Spring 2016
        - OpenDSA HW Assignment 3, Stacks
        - Lab 06: Pre-lab Activity Testing
        - Lab 06: ArrayBasedStack
        - Lab 06: Post-Lab Activity UML Diagrams (Not Graded)
        - Week 6 practice problems
        - CS2114 Lecture 08 Stack Implementations
        - Project 3: Tower of Hanoi
  - My courses

SETTINGS

- Course administration
- My profile settings

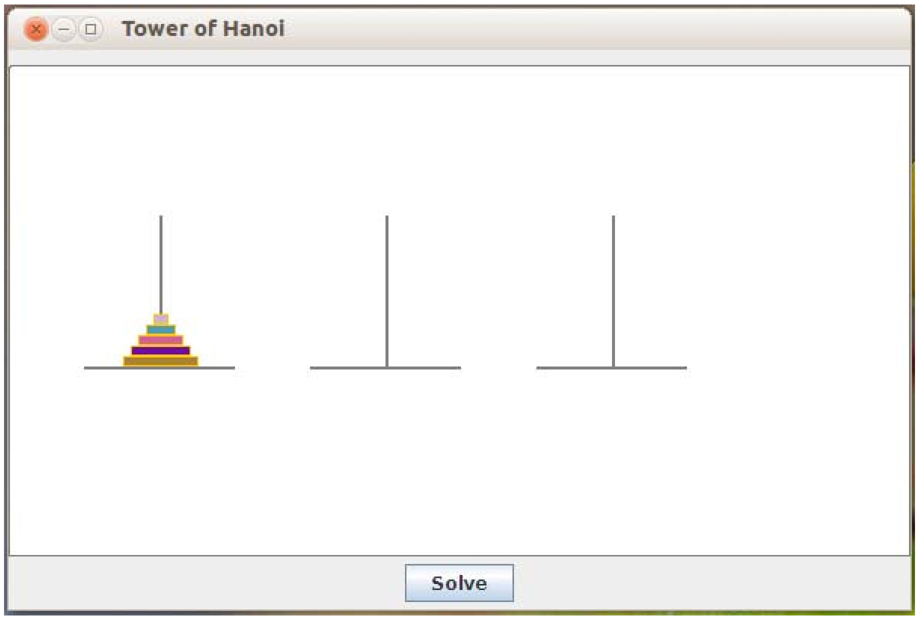
PROJECT 3: TOWER OF HANOI

Project 3: Tower of Hanoi

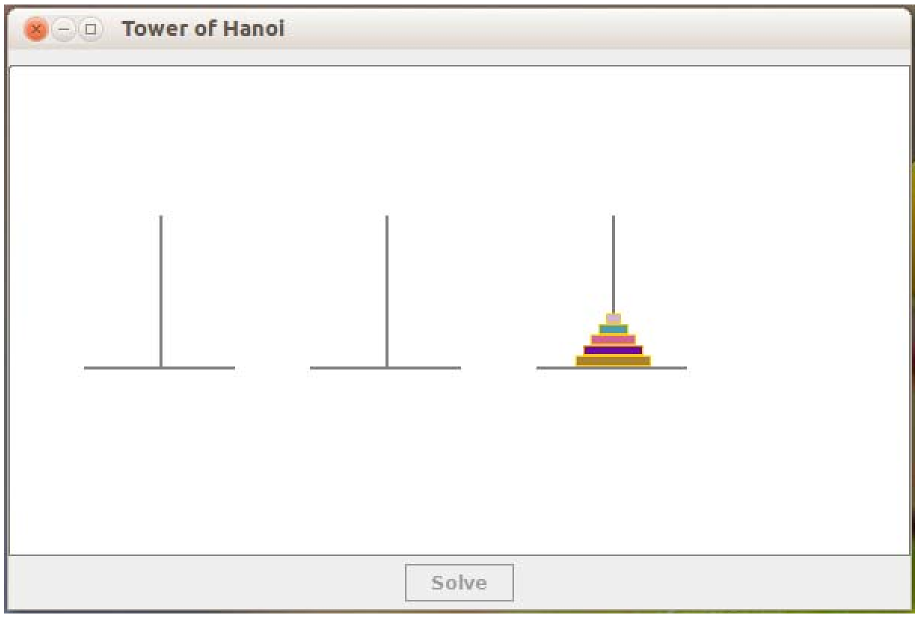
Due Date: Thursday, March 17. 8am.

Introduction

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 textbook, you will solve a game, step by step, and display your moves in a `Window`. You'll start the game with the program display looking something like this...

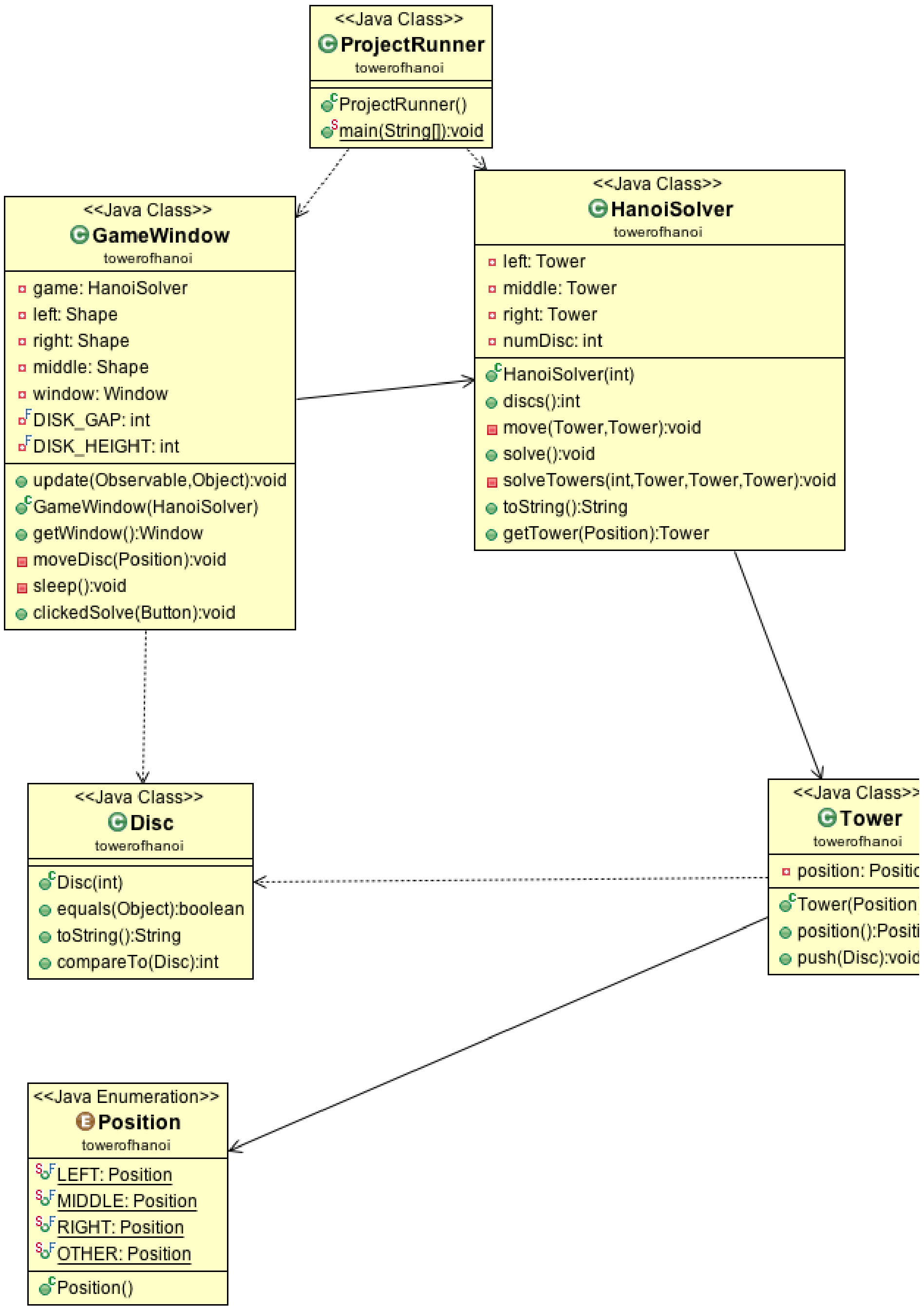


and end it looking something similar to:



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

UML



Implementation Tips

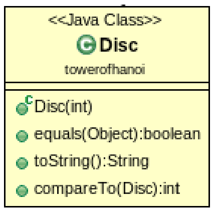
There are several classes that throw Runtime exceptions. Because these are Runtime exceptions and not checked exceptions, a throws clause is not necessary. However, it is recommended that conditions that cause the Runtime exceptions be listed in the class's JavaDocs with @precondition tags.

Create your project 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 `LinkedStack` from CarranoDataStructuresLib 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.

Disc extends Shape implements Comparable<Disc>

The `Disc` class extends `Shape` because we are thinking of Discs as rectangles of various widths. (For a reminder about how `Shape` works, check the API.) Dimensions are important for determining valid moves, since only smaller discs can be placed on larger ones. `Disc` also implements the `Comparable<Disc>` interface to allow Discs to be compared to one another.



Disc(int width)

Call the `super()` constructor, which takes 4 pararemeters: x, y, width and height. Use coordinates (0,0) for now, and a width and a height of your choosing. Between 5 and 10 is recommended. We'll move these discs after they're built. After calling super, set the disc's background color to a random new `Color`. To randomly generate a color, call the `Color` constructor with three random integers ranging from 0 to 255.

compareTo(Disc otherDisc)

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

toString()

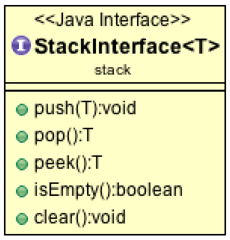
Return the width of this `Disc` as a string. Use its `getWidth()` method. For example, calling `toString()` on a disc of width 10, `disc1.toString()`, should return "10".

equals(Object obj)

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

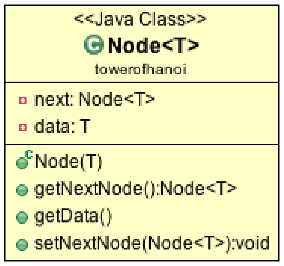
LinkedStack implements StackInterface

Write a `LinkedStack` class that implements the given `StackInterface`. You can view details about `StackInterface` in the API. Implement this stack with linked nodes. Include a size field and the size() method for keeping track of how many objects are in the `LinkedStack`. The front end of our project will need this information from the backend. Be sure your project is using your LinkedStack implementation and not the one from CarranoDataStructuresLib.



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 textbook). Be sure your project is using your Node implementation and not the one from CarranoDataStructuresLib.



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 an instance of `LinkedStack`, `stack1.toString()` should output "[lastPush, secondPush, firstPush]", or "[]" if `stack1` is empty. Notice the order of the output is from top to bottom.

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.

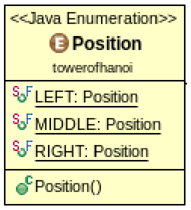
toString() make toString() look the same as if you were printing out an array with the same contents. It is generally preferable to use a StringBuilder for string concatenation in Java. It is much faster and consumes less memory.

enumerator Position

There are always three poles in the Towers of Hanoi game, 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…

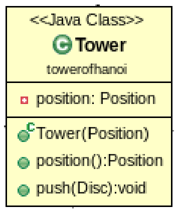
```
LEFT, MIDDLE, RIGHT, OTHER;
```

… and will be used to determine the `Tower` position.



Tower extends LinkedStack<Disc>

The `Towers` on which we store `Discs` function as stacks. We extend the `LinkedStack` we just implemented, since Towers offer a unique extension to a normal stack – they only allow smaller discs 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.

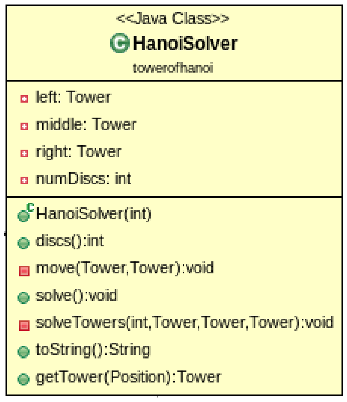
position() Return your `Tower`'s position here.

push(Disc disc) 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 disc provided. If this tower is already empty, or the disc on top is larger than the disc being pushed, we have a valid push. Use the disc's `compareTo()` method to determine which is smaller. If it is a valid push, we call `super.push()`, and provide the disc. If this is an invalid push, we will throw an `IllegalStateException`.

HanoiSolver extends Observable

Your `HanoiSolver` represents a Tower of Hanoi game. These games vary in regard to how many discs 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`, so that the `GameWindow` may observe `HanoiSolver` to update the display which animates the Discs.



HanoiSolver(int numDiscs)

Every Tower of Hanoi game requires the number of discs to be specified. Store numDiscs 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.

discs()

return the number of discs, numDiscs.

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 other. 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 left tower.

toString()

Return left , middle, and right's `toString()` appended. For example: if the left, middle, right tower each have a single disc 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 discs onto its towers.

move(Tower source, Tower destination)

This method executes the specified move. Pop the `Disc` 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 tells any observers that it's time to update with a call `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 currentDiscs, Tower startPole, Tower tempPole, Tower endPole)

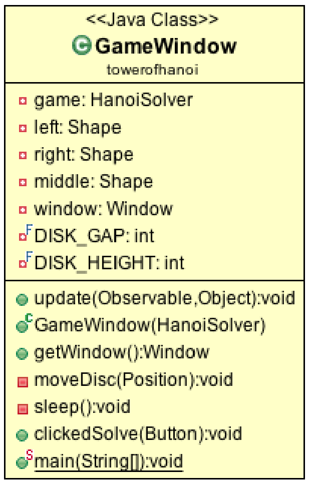
Implement your recursive `solveTowers()` method with help from your textbook. Start with the base case where there is only one disc left to move. Otherwise, you recursively solve smaller subproblems by calling `solveTowers()` with slightly different parameters, invoking the `move()` method when it is necessary for a disc to be moved.

solve()

Your `solve()` method will be public and it makes the initial call to the recursive `solveTowers()` method. It provides the `solveTowers()` method the correct parameters, with left being the start, middle being the temp, and right being the end.

GameWindow implements Observer

Our `GameWindow` creates the `Window` in which we view the game, and observes `HanoiSolver`given to it by the main method for updates on when to animate the `Window`. Because this class is an `Observer`, it requires the `update()` method. Leave it empty for now. This front end class should have one `HanoiSolver` field named game, and three `Shape` fields indicating the left, middle and right towers on the front end. How you use and name your constants is up to you, we grade for use of conatants but since the ones in this method are private WebCAT will not be checking for specific usage.



methods using Threads

Copy and paste the two methods below into this `GameWindow`. They deal with making a `Thread`, which you don't have to learn (yet!). The `sleep()` method we will use to pause between Disc 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 backend changes. Try to understand how they work, and read a little about Threads online.

```
private void sleep() {
    try {
        Thread.sleep(500);
    } catch (Exception e) {
    }
} // end sleep

public void clickedSolve(Button button) {
    button.disable();
    new Thread() {
        public void run() {
            game.solve();
        }
    }.start();
}
```

moveDisc(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 `Disc` that was just moved to get needed information for the display. The backend already moved the discs between the Towers.

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

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 Disc inherits from `Shape` will graphically relocate the disc in the display. Use the `getX()`, `getY()`, `getWidth()`, and `getHeight()` methods for the disc and pole to determine the new X and Y coordinates to which the disc will move. You will also need the `size()` of the tower to know how many discs are underneath the current one, to adjust its Y position appropriately.

GameWindow(HanoiSolver game)

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.

Declare a new Window as a class field. Give it the title "Tower of Hanoi".

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.

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

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

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 SOUTH side of the Window, and tell it to `onClick(this)`.

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 passed as our `arg` parameter here.

To start, check if `arg.getClass()` equals `Position.class`. If true, cast your arg to be a `Position`. Call the previously created `moveDisc` method with your position, then call `sleep()`.

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

ProjectRunner

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

main(String[] args)

The last method! Here, we make a new `GameWindow`, and pass it a new `HanoiSolver`, which we pass the number of discs to use. To determine how many discs to use, declare a local variable named discs which is five by default. If the `String` array `args` length is exactly one, set your discs integer to be the Integer parsed from the `args[0]` location (`discs = 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.

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

Optional: Extra 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 discs 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 games, adjust your sleep time based on the number of discs. Modify the provided `sleep()` method to provide this behind-the-scenes tweak.

Allow user interaction with discs

To let users solve the game themselves, you can allow users to drag and drop their shapes. Provide the shapes with an onClick method, such that users can click a Disc, 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 discs bigger than in the examples to make them easier to click on.

Submission status

Submission status	This assignment does not require you to submit anything online
Grading status	Not graded

Due date	Thursday, March 17, 2016, 8:00 AM
Time remaining	The due date for this assignment has now passed

You are logged in as Mykayla Fernandes (Logout)

CS 2114 Spr 2016