|  |
| --- |
| [CS](https://www.plymouth.edu/department/computer-science/) 2381: Data Structures  (Fall 2020) |
| [**Syllabus**](https://turing.plymouth.edu/~kgb1013/?course=2381)  [**Moodle**](http://www.plymouth.edu/courses/course/view.php?id=51851)  **Teachers**  [Kyle](https://turing.plymouth.edu/~kgb1013/)  **Assignments**  [Project 0](https://turing.plymouth.edu/~kgb1013/?course=2381&project=0)  [Project 1](https://turing.plymouth.edu/~kgb1013/?course=2381&project=1)  [Project 2](https://turing.plymouth.edu/~kgb1013/?course=2381&project=2)  [Project 3](https://turing.plymouth.edu/~kgb1013/?course=2381&project=3)  [Project 4](https://turing.plymouth.edu/~kgb1013/?course=2381&project=4)  [Project 5](https://turing.plymouth.edu/~kgb1013/?course=2381&project=5)  [Project 6](https://turing.plymouth.edu/~kgb1013/?course=2381&project=6)  [Project 7](https://turing.plymouth.edu/~kgb1013/?course=2381&project=7)  [Project 8](https://turing.plymouth.edu/~kgb1013/?course=2381&project=8)  **Other Pages**  [Kyle's Teaching](https://turing.plymouth.edu/~kgb1013/?main=teaching) [Kyle's Schedule](https://turing.plymouth.edu/~kgb1013/?main=schedule) [Kyle's Resources](https://turing.plymouth.edu/~kgb1013/?main=resources) | **Project 3: The Stacks are High**  **Assigned: Tue Sep 22 2020 Due: 11:59:00 PM on Mon Oct 05 2020 Team Size: 1 or 2 Language: Java Out of: 100 points**  In this project, you will implement the stack data structure and use it to create a player for Tower Nim. Since we are only going to implement very basic stack operations, we'll name the class PureStack. In this project, you are not allowed to use any pre-defined Java classes that already implement stack functionality (such as java.util.Stack.)  **Part 0, 5 points:**Start off by creating a new folder for this project. Then create a new file, PureStack.java. Feel free to use my [EmptyClass.java](https://turing.plymouth.edu/~kgb1013/DB/code/EmptyClass.java).  **Part 1, 0 points:**Next, set the class signature for PureStack. We're going to use generic types, so that your stack can hold objects of any single type. Thus, the header should look something like this:  public class PureStack<E extends Object> {  **Part 2, 10 points:**Next, implement a constructor that takes no arguments. Write the Javadoc. What sort of field(s) will your stack use? (Remember, you can't use the java.util.Stack class!) I highly recommend building off of another data structure we've used in lab ([wink, wink](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html)). After you've written the Javadoc, write the constructor.  **Part 3, 10 points:**Write a toString method. (Start by writing the Javadoc, naturally.) My toString prints out all the elements and indicates at which end the top is. I use mine like the following: (You haven't written push yet, so you can't run this entire test yet.)  PureStack<String> stringStack = new PureStack<String>();  stringStack.push("Hi");  stringStack.push("yo");  stringStack.push("Animal!");  System.out.println(stringStack);  The last line prints out something like this:  Top --> [Animal!, yo, Hi] <-- Bottom  or this:  Bottom --> [Hi, yo, Animal!] <-- Top  **Part 4, 0 points:**Start your unit test and print out an empty stack to make sure everything's working as you want it to.  **Part 5, 10 points:**Implement push. (Javadoc first!) This should be a void method that takes one parameter of type E and adds it on top of your stack. Add it to your unit test. Test your code thoroughly!  **Part 6, 10 points:**Implement pop. This method could throw a NoSuchElementException. That line of code will look something like this:  throw new NoSuchElementException("Use a better string than this.");  When should your code throw the exception? When you have an idea, add the appropriate tag to your Javadoc, which will look something like this:  \* @throws NoSuchElementException When blue martians take over this code.  Code up your solution. You can test that your code works by catching the exception in your unit test, doing something like this:  PureStack<Alpaca> alpacas = new PureStack<Alpaca>();  try {  alpacas.pop();  //if the code reaches this point, the exception didn't get thrown!  isCorrect = false;  } catch (NoSuchElementException e) {  //the exception was thrown; the method worked correctly!  isCorrect = true;  }  System.out.println((isCorrect ? success : failure));  allCorrect = allCorrect && isCorrect;  I found a little [exception tutorial](http://www.tutorialspoint.com/java/java_exceptions.htm) in case you'd like more background on exceptions. (I can also answer any questions you have about this, of course!) Hint to finishing the method: in the code, there are two ways to test for when you should throw your exception: either use a conditional or catch another possible exception that would be thrown.  **Part 7, 0 points:**Put pop through another gauntlet of tests in your unit test to make sure it works correctly. Try to consider all possible cases that could have very different results in your code. I would recommend at least the following cases:   * Call it on a stack with no elements. (This should throw the exception, which you'll want to handle in your unit test.) * Call it on a stack with one element. (It should be empty afterwards.) * Call it on a stack with two elements. * Call it on a stack, then put an element back into the stack and make sure it has that new element.   (From here on out, I'll expect you to do your own unit testing and won't remind you about it.)  **Part 8, 10 points:**Implement peek. Consider: should this ever throw an exception? Once you figure that out, there's a really elegant way to do this using methods you've already written. (At this point, I'll stop reminding you to first write the Javadoc.)  **Part 9, 10 points:**Implement isEmpty. This should take no arguments and return a boolean.  **Part 10, 10 points:**We need to be able to test whether two Stacks are equivalent. Next, we'll implement equals. Just as in previous projects, we need to at least have a version of this that takes an Object parameter. Take a good look at how you did this in the Pair project and follow that plan. Extra Hint: Take a good look at the methods available to your field. I think you can write this method in one line!  **Part 11, 0 points:**Let's test your code out during actual game play. You'll need some things:   * Abstract CG class: download and compile [CombinatorialGame.java](https://turing.plymouth.edu/~kgb1013/DB/2381/code/AllProjects/CombinatorialGame.java). * Player classes: [Player.java](https://turing.plymouth.edu/~kgb1013/DB/2381/code/AllProjects/Player.java) and [RandomPlayer.class](https://turing.plymouth.edu/~kgb1013/DB/2381/code/AllProjects/RandomPlayer.class). * Abstract class for generation of game states: [PositionFactory.java](https://turing.plymouth.edu/~kgb1013/DB/2381/code/AllProjects/PositionFactory.java). * Referee: [Referee.java](https://raw.githubusercontent.com/paithan/CombinatorialGameCode/master/java/Referee.java).   **Part 12, 0 points:**Now download the [TowerNim.java](https://turing.plymouth.edu/~kgb1013/DB/2381/code/StackProject/TowerNim.java) source file and put it in your folder. Read through the main method in TowerNim.java. I think my unit test is quite thorough, though feel free to let me know if you would suggest other tests! Try compiling and running this with your PureStack implementation. Does the unit test pass or fail? If there are issues, there's probably a problem with your code. (There could be a problem with mine; I'm not perfect!) Fix all the errors before continuing.  **Part 13, 0 points:**Create another new class, TowerNimPlayerTester.java, which you'll use to test your player. You will not need a constructor, because you will not be creating instances of this class. All you need is a main method which creates the players, creates the referee, and runs the code. Put these lines in your main method:  Player<TowerNim> random = new RandomPlayer<TowerNim>();  PositionFactory<TowerNim> factory = new TowerNim.PositionBuilder(8, 8);  Referee ref = new Referee(random, random, factory);  ref.call();  Try compiling and running the code. (We are using a new class so that we can leave the main method of Referee.java as a unit test.)  **Part 14, 0 points:**Play a few games of Tower Nim with your team. What are some aspects of good strategies here?  **Part 15, 15 points:**Create your own Tower Nim player class, TowerNimPlayer.java. The code for this will look very similar to the player for the last project, except you'll be dealing with TowerNim objects. When using those objects, invoke only the methods I asked you to write in this project (I will test with my version of the data structure, so using extra methods will break your player). Your code should not make use of any of the players I've provided you with. Make sure your player compiles, then write a unit test and ensure it chooses moves in a reasonable amount of time without any forfeits. Remember:   * Your player should only directly invoke the PureStack methods assigned here. I'll be testing your player with my own copy of PureStack.java, so if you call other methods, I won't be able to test your player. * Don't use randomness in your player. (Randomness is a really powerful tool. If you're interested in writing a player that uses randomness, we should definitely talk after this course is finished!) * Don't call the getOptions method.   **Part 16, 0 points:**Test the awesomeness of your player by modifying your player tester class to test with my random player.  **Part 17, 10 points:**Improve your player so that it consistently beats my random player:   * 35% of the time or more: 5 points * 50+%: 10 points * Random player 65+%: 15 points (5 is bonus)   **Part 18, 10 points (Bonus):**Download the [PerfectTowerNimPlayer](https://turing.plymouth.edu/~kgb1013/DB/2381/code/StackProject/TowerNimPerfectPlayer.class) class file and put it in the same folder. Change your testing class to use this new player. How often does your player win? Tweak your player so that it consistently wins at least 48% of the time. (Warning: you must beat 48% using the settings I choose.) If you do this, you've probably written a perfect player! Nice job!  **Submitting your Project:**  Be careful to follow all these directions precisely in order to ensure I can grade your assignment in a timely manner. If you don't, I may give you zero points for the assignment.  Make sure your code all compiles from the command line:  javac \*.java  Make sure your player class doesn't use any methods of the data structure except for the ones I asked you to write. (I will be testing your player with my own version of the data structure. That way if there are any issues with your data structure you didn't notice, your player can still run correctly.)  Please check that your code only prints to the screen in the tests and not when being tested by the Referee. If it does, please comment those print statements out.  Create a *new* folder to submit your files in.   * If you are working alone, Name this folder *YourLastName*Project3, all in PascalCase. (For example, my folder name would be: BurkeProject3.) * If you are working in a group with two people, create a new folder labelled with your two last names, separated by "And", and followed by Project3 all in PascalCase. (For example, if I worked with Kathleen Stock, our folder would be named BurkeAndStockProject3.)   Copy your source code (.java files) into that directory. Your code should be at the top level of that directory (not in any subdirectory). You do not need to submit any extra classes you used to test your structure or player. (I've got really tough tests to do that myself!) I'll be looking for the following files in your folder:   * PureStack.java * TowerNimPlayer.java   Finally, compress your file into a .zip file. (Don't use WinRar!) Upload the zipped file to the project page on Moodle. If you're submitting late, please also send me an email (without the code attached) so I know there's something waiting for me on Moodle. |