|  |  |
| --- | --- |
| Mary Lyon Snow | [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 0: Pairing Up**  **Assigned: Mon Aug 24 2020 Due: 11:59:00 PM on Wed Sep 09 2020 Team Size: 1 Language: Java Out of: 100 points**  This project is a basic walkthrough of a basic data structure which holds a pair of elements. We'll see generic types, unit tests, field accessibility, and more! Afterwards, we'll play an abstract game that uses Pairs; you will write code to choose a move from any position. This is the usual pattern for projects in this course: first we'll program the relevant data structure, then play a game based off of it. If you don't feel fully comfortable with Java, at any point you can complete my [From Python to Java Tutorial](http://turing.plymouth.edu/~kgb1013/pythonToJavaTutorial/start.php). Some people like to do that before this, and some people find it more helpful to do the tutorial after completing this project.  **Part 0, 0 points:**Let's run through a quick Windows command-line refresher! Click on the Start button, and type cmd into the little box, then hit enter. Move into the Z drive by typing this:  > Z:  Next, create a folder for our class (if you haven't already) with the following command. (You can use a different name.)  > mkdir 2381IsAwesome  Move into that directory with  > cd 2381IsAwesome  Create a subfolder and move into it like so:  > mkdir project0 > cd project0  Keep this command window open so you can use it to compile and run your Java code.  **Part 1, 0 points:**Open your favorite text editor (or development environment) and start a new file. Immediately save it as Pair.java inside your project0 folder. This will be where you'll put the code for the Pair data structure we're building. Let's start by adding the Javadoc header to your project. Add the following text to your file:  /\*\*  \* Represents a Pair of values.  \*  \* @author YourNamesHere  \*/  **Part 2, 0 points:**Now we need to add the actual class declaration. For now, let's just leave everything inside empty. Add the following after your header:  public class Pair {  //code will go in here  }  **Part 3, 0 points:**At this point, we should be able to compile and run the Java source code. In case you've forgotten how to do that via command-line, use these two lines:  > javac \*.java > java Pair  The first line compiles the code into Java Bytecode (the .class files that are created). The second line runs the main method in the Pair class. Since we don't have a main method in that class yet, you should have gotten an error.  **Part 4, 0 points:**We can, however, compile the Javadoc. To do this, use  > javadoc \*.java  This generates a bunch of webpages inside the folder that you (and others) can use to read comments about the code. The generated pages look just like the online Java API. These pages are the API for the code you've written, and can be referenced by programmers who want to use your code. Take a few minutes to find the page generated by the code you just ran. As we add more Javadoc to our code (and recompile the Javadoc) these pages will grow! (Yes, I will be expecting you to use Javadoc correctly throughout the semester.)  **Part 5, 0 points:**Let's actually add some functionality to our code. First: a constructor! Add the following code inside your class: (inside the open and close squiggles)  public Pair(Object first, Object second) {  this.first = first;  this.second = second;  }  Anytime you open a squiggly-brace, the following lines should be indented (I like four spaces) until you reach the corresponding close-squiggly. Thus, your overall code (not counting the Javadoc header) should now look like this:  public class Pair {  public Pair(Object first, Object second) {  this.first = first;  this.second = second;  }  }  **Part 6, 0 points:**Hopefully you are either upset or confused right now. There are so many things wrong! Try compiling this code and you should get a bunch of errors! The first problem is with the assignment this.first = first;This line won't compile because there is no declaration for the field named first. In order to declare the field, put the following above the constructor:  //the first element in the pair  private Object first;  This says two things about the 'first' field:   * The field is private, meaning it can't be accessed by any other classes. * Any value assigned to this field will be an Object.   Where should this line be indented? It's inside the class, but outside the constructor, so it should be flush with the constructor's signature:  public class Pair {  //the first element in the pair  private Object first;  public Pair(Object first, Object second) {  ...  **Part 7, 0 points:**The other thing you might be concerned about is that the field has the same name as the constructor parameter! If a parameter ever has the same name as a field, you can still specify the field by including the this. prefix. Leaving this prefix off refers to the parameter. Consider the following example:  public void methodName(Object asdf) {  this.asdf = 5; //sets the field  System.out.println(asdf); //prints parameter  System.out.println(this.asdf); //prints the field  }  With most methods, it's a terrible idea to have parameters with the same name as fields, but it's common to do with the constructors if the field is going to be assigned directly to the parameter.  **Part 8, 0 points:**We've added a field for first. Now do the same for second. When you've finished, your code should compile.  **Part 9, 20 points:**Fields, methods, and constructors (and inner classes) in a class are all members of that class. Whenever we make a new public member, we use Javadoc to add a description of it for use by others. Add the following Javadoc header above your constructor:  /\*\*  \* Class constructor.  \*  \* @param first The first element in this Pair.  \* @param second The second element in this Pair.  \*/  public Pair(...  The @param tags describe the parameter. Whenever you have one of these tags, it should be followed first by the parameter name, then a sentence (or two) description of what role the parameter plays in the outcome of the method. Recompile the Javadoc and look at the new page that's generated.  **Part 10, 0 points:**Now that we have compiling code, we can test the code to make sure it works. Let's start a unit test! Add the following after the constructor:  ...  }  /\*\*  \* Unit test.  \*  \* @param args Command-line parameters for this test. Currently unused.  \*/  public static void main(String[] args) {  Pair twoThings = new Pair("Hi", new Integer(5));  System.out.println("Test completed successfully!");  }  Recompile the code, and then run it. If it works, you should see the following:  > javac \*.java > java Pair Test completed successfully! >  Recompile the Javadoc to see the new method entry for the unit test in your API. In general, your Javadoc method headers should have this format:  /\*\*  \* Brief description of what the method does (but not how it does it).  \*  \* @param x Description of parameter x's role in the method.  \* ... other  \* tags  \* here...  \*/  **Part 11, 0 points:**All that our successful unit test tells us is that the constructor doesn't throw any exceptions. We'll be writing lots more methods, so it'll be helpful to know what the state of the Pair object is between those method calls. Then we can properly check that those methods work correctly! Whenever you create a new class, it's a good idea to write a toString method which takes no parameters returns a String that describes the object's state. We'll have Pair's toString return a String formatted in parentheses with a comma between the elements. It's common practice to keep the unit test at the bottom of the class, so add the following code to your program above the main method:  /\*\*  \* Returns a String representation of this Pair.  \*  \* @return A String, "(X, Y)", where X is the first element and Y is the second element.  \*/  public String toString() {  return "(" + this.first.toString() + ", " + this.second.toString() + ")";  }  **Part 12, 0 points:**Notice that we added a new tag here! Try recompiling the Javadoc and look at the section for the new method. What's different? Why don't we have to specify a variable name?  **Part 13, 0 points:**Make sure your code compiles. What should we do now to make sure our toString is behaving properly? Yes! Test it in our unit test! One way to do this is to print out the results and verify it visually. Change your unit test (main method's body) to the following:  Pair twoThings = new Pair("Hi", new Integer(5));  System.out.println("twoThings: " + twoThings.toString());  System.out.println("Test completed successfully!");  Compile and run the code. You should see this as part of your output:  twoThings: (Hi, 5)  **Part 14, 0 points:**The toString method in Java is special: it automatically gets called on non-string objects in an expression where a string is needed. Change the unit test by getting rid of the toString invocations. (E.g. change twoThings.toString() to just twoThings .) Rerun your unit test. Does the result change at all?  **Part 15, 10 points:**Your own toString method calls the toString methods of the fields. Change this like the unit test by removing the actual invocations, since they'll be called anyways. Retest your main method to make sure everything's still working.  **Part 16, 0 points:**I like to make the computer do things for me. We can do this by modifying the unit test to do some string comparisons. Modify the body of your unit test to look like this:  String success = "Test completed successfully!";  String failure = "Something went wrong!!!!!!!!!!!!";  Pair twoThings = new Pair("Hi", new Integer(5));  String output = twoThings.toString();  System.out.println("twoThings: " + output);  boolean isCorrect = output.equals("(Hi, 5)");  System.out.println((isCorrect ? success : failure));  In case you're not familiar with this syntax, X ? Y : Z is Java shorthand for a sort of in-line conditional. It evaluates the boolean expression X, then   * If true, the whole thing evaluates to Y. * If false, the whole thing evaluates to Z.   Compile and run your class. You should see the success message when it runs.  **Part 17, 0 points:**There is still lots of functionality we can add. It's very natural to want to get the elements so we can use them. Let's write a new method getFirst to get the first one. This sort of method is known as an accessor (or more colloquially as a "getter"). Getters return a part of the object without modifying it in any way. Since we have a really good idea of what this method will do, we can write the Javadoc before writing the actual code:  /\*\*  \* Returns the first element.  \*  \* @return The first element in this.  \*/  When writing a new method, it's a good idea to try to write the documentation first. If you don't know what the purpose of the method will be, you'll probably have trouble writing the method itself!  **Part 18, 0 points:**Now let's write the method signature and body. This is an easy one-line method:  public Object getFirst() {  return this.first;  }  **Part 19, 0 points:**Let's test this animal! Change your unit test like so:  String success = "Test completed successfully!";  String failure = "Something went wrong!!!!!!!!!!!!";  Pair twoThings = new Pair("Hi", new Integer(5));  String output = twoThings.toString();  System.out.println("twoThings: " + output);  boolean isCorrect = output.equals("(Hi, 5)");  System.out.println((isCorrect ? success : failure));  output = (String) twoThings.getFirst();  isCorrect = output.equals("Hi");  System.out.println((isCorrect ? success : failure));  //further tests will go here.  It's important that we don't remove old tests from the code. When you add new parts to your unit test, leave the old parts. That way you can make sure old stuff is still working correctly after you make new changes! Run your code and make sure you only get success messages.  **Part 20, 0 points:**We can use the program to test and make sure that all parts are working correctly by introducing a new variable. Add the following line to the top of your unit test:  boolean allCorrect = true;  Then, after each assignment to isCorrect, I added this line:  allCorrect = allCorrect && isCorrect;  Now, if one of the tests ever fails, allCorrect will be set to false and will stay that way through the whole test. To finish this off, I added one line to the end of my test to tell me whether I had any failures:  //further tests will go here.  System.out.println((allCorrect ? "All tests were successful!" : "At least one test failed!"));  Now you only have one place to look to make sure everything worked. (If something did fail, you should be able to spot it easily!)  **Part 21, 0 points:**What happens if we try it without the type cast, like in the following line? (Actually try this out. What happens?)  output = twoThings.getFirst();  We can fix this by using generic types in Java. This will allow us to specify the type of the first element when we declare the variable. Let's do a quick run-through of how to do this! First, we need to change the class signature to include a type variable. Change it to:  public class Pair<FirstType> {  Now we need to change the field so that it uses that type throughout the program. Change the field declaration:  private FirstType first;  Change the type of the parameter in the constructor:  public Pair(FirstType first, Object second) {  And change the return type in the getter:  public FirstType getFirst() {  Finally, we need to change how the class is used. In your unit test, modify the line where you create the pair:  Pair<String> twoThings = new Pair<String>("Hi", new Integer(5));  **Part 22, 0 points:**We can use this syntax to create pairs with other types of first elements. Add the following to your unit test to create a pair with an Integer:  Pair<Integer> pair = new Pair<Integer>(6, "Hi");  It's important to be thorough with unit tests. If you test a method in only one way, then you're likely to miss a lot of cases that could have errors. Be thorough! To this end, let's add a new piece to our unit test:  Integer integer = pair.getFirst();  isCorrect = integer.equals(6);  System.out.println((isCorrect ? success : failure));  allCorrect = allCorrect && isCorrect;  Rerun your test to confirm that everything's working.  **Part 23, 0 points:**For most data strucutres, we'll want to use one generic type for all elements. We could do that here by using the same generic type for both elements. I'd rather show you something a bit different and make our pair more flexible. So, let's use a second type variable for the other element. The first step is to change the class signature again:  public class Pair<FirstType, SecondType> {  Now go through and change the second element to use that type. Update:   * The field declaration. * The constructor signature. * The constructor calls in the unit test.   For this last one, you now need to include the other type variable whenever you're creating a new pair, so using the constructor will look like:  Pair<String, Integer> things = new Pair<String, Integer>("Beluga", 56);  As always, rerun your unit test.  **Part 24, 20 points:**Let's add a getter for the second field. Create a method, getSecond, which has signature  public SecondType getSecond() {  Add a couple method invocations to your unit test to check that it works correctly. You'll need to use a new (Integer) variable for this since output is a string. I was really boring and did this:  Integer integer = twoThings.getSecond();  **Part 25, 0 points:**Now that our getters are written, let's add some setters. Setters are void methods (they don't return anything) that set a field to a given value. For our first setter, let's change the first element. It would be great to have users invoke our method like so:  Pair<Camel, Llama> camelids = new Pair<Camel, Llama>(...);  ...  camelids.setFirst(new Camel("Daniel", 2));  (I know you don't have either of those two classes, but I wanted to make it clear that the elements can have any type, including ones you might create!) Let's write the Javadoc first:  /\*\*  \* Sets the first element.  \*  \* @param newFirst The value to set the first element to.  \*/  Then add the method directly beneath it:  \*/  public void setFirst(FirstType newFirst) {  this.first = newFirst;  }  I used the following test to check that my method worked:  pair.setFirst(-124);  integer = pair.getFirst();  isCorrect = integer.equals(-124);  System.out.println((isCorrect ? success : failure));  allCorrect = allCorrect && isCorrect;  Is this test enough? Not really! Add another test, this time with a pair that has a string in the first element.  **Part 26, 10 points:**Write the Javadoc for a setter, setSecond, then write the method. Make sure it works by adding code to your unit test. (Remember not to remove the old tests!)  **Part 27, 0 points:**It's going to be important to test whether two pairs are equivalent to each other. We'll do this by overriding the Object class's equals method. Let's actually start this one out by writing the unit test first so you can see what I'm shooting for. Add the following code to your unit test:  pair = new Pair<Integer, String>(16, "beluga");  Pair<Integer,String> otherPair = new Pair<Integer, String>(16, "beluga");  System.out.println("Now testing whether " + pair + " equals " + otherPair + ":");  isCorrect = pair.equals(otherPair);  System.out.println(isCorrect);  System.out.println((isCorrect ? success : failure));  allCorrect = allCorrect && isCorrect;  Now test your code. What happens? Should the two pairs be equivalent? This test fails because the Object class tests equivalence by just comparing the memory addresses of the two values, not by looking at the actual data.) Since we don't have an equals method, our class inherits the method from it's superclass. (Since we didn't specify a different superclass, Object is the automatic superclass.) In order to correctly test equivalence, we need to write our own equals method to override the one from the Object class. Add the following Javadoc for this method:  /\*\*  \* Tests whether two Pairs are equal.  \*  \* @param other Another pair that might be equivalent to this.  \* @return True if both the first element of this equals the first element of other  and the second element of this equals the second element of other, false otherwise.  \*/  Notice the difference between the brief explanation of what the function does and the more thorough explanation in the @return tag. Most of the time it will be clear what this method does by reading the brief description, but if a deeper description is needed, that's given by the @return tag.  **Part 28, 0 points:**Let's add the code for the method:  public boolean equals(Pair<FirstType, SecondType> other) {  boolean firstsEqual = this.getFirst().equals(other.getFirst());  boolean secondsEqual = this.getSecond().equals(other.getSecond());  return firstsEqual && secondsEqual;  }  Add the following checks to your unit test:   * Check two integer-string pairs with the same values. (Should be true.) * Check two integer-string pairs with different values. (Should be false.) * Check two string-string pairs with the same values. (true) * Two string-string pairs with different values (false). * One integer-integer pair and one integer-string pair. (Should be false.)   Can this last test ever be true? Nope! In fact, it won't even call your method because the type of the parameter doesn't match up (the generic types are different). Instead, the Object class's version of the method is called. Unfortunately, the same thing happens in this case with an added typecast:  pair = new Pair<Integer, String>(16, "bebop");  otherPair = new Pair<Integer, String>(16, "bebop");  isCorrect = pair.equals((Object) otherPair);  System.out. ...  Finish filling in the details for this test, then run your code. What happens? Which class's version of equals is getting called? Why is this happening?  **Part 29, 0 points:**The reason all of this is happening is because we haven't fully overriden Object's equals method. If you look at the [API for Object.equals](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-), the signature is actually  public boolean equals(Object obj)  In order to override this, we need to write a version of the method with the exact same signature. Let's do this now:  /\*\*  \* Returns whether this equals another object.  \*  \* @param obj The object to determine equivalence with this.  \* @return True if obj is a Pair with elements equivalent to this, false otherwise.  \*/  public boolean equals(Object obj) {  try {  Pair<FirstType, SecondType> other = (Pair<FirstType, SecondType>) obj;  return this.equals(other);  } catch (ClassCastException e) {  return false;  }  }  Don't be frightened if you've never seen an exception-handling block before! Let's take a closer look at what's going on. The try keyword tells me that something inside the following squiggly braces might cause an exception (a run-time error). The catch (ClassCastException e) block afterwards is the code that will get called in case a ClassCastException is thrown. Here, that will happen if obj isn't something that's actually a Pair<FirstType, SecondType>. In that case, the casting line will fail and the thread of execution will skip to the code inside the catch block, which just returns false. Rerun your unit test and you'll see that the last case succeeds! Great!  **Part 30, 0 points:**What happens if we try that test with a cast the other way around? Add another piece to your unit test:  isCorrect = ((Object) pair).equals(otherPair);  System.out.println ...  Finish filling in that test and try running your code. Does it invoke the correct version of equals? It should, because Java uses polymorphism, which means that while running, Java checks the actual type of the subject value (the subject is the object that methods are invoked on, the thing before the dot) and uses any overriding versions of the methods. So even though the subject is declared to be of type Object, Java double-checks at run time and notices that the actual type is Pair. This functionality is called Dynamic Method Invocation. You might have noticed when compiling your code that you now get a message:  > javac \*.java Note: Some input files use unchecked or unsafe operations. Note: Recompile with -Xlint:unchecked for details. >  The problem here is with the typecast you have to do when using Object, but the actual type of the value is String in this case. The compiler doesn't know this, though.  **Part 31, 30 points:**We're nearly done with the methods for this class. Let's write one more: getReverse, which will take no arguments and returns a new pair with the elements reversed. Here's the code I used:  /\*\*  \* Returns a pair with the elements swapped.  \*  \* @return A new Pair with the elements in reverse order.  \*/  public Pair<SecondType, FirstType> getReverse() {  return new Pair<SecondType, FirstType>(this.getSecond(), this.getFirst());  }  Write a few tests to make sure that this works.  **Part 32, 0 points:**You've completed the part of the project where you create the data structure. For most projects, this will usually earn you somewhere between 80 - 90% of the points if you do everything perfectly. The rest of the project is spent working with playing an abstract board game (more specifically, a combinatorial game) that uses the data structure heavily. For pairs, that game is Wythoff's Nim. In this game there are two piles of sticks, represented by a pair of integers. Each turn a player can do one of the following moves:   * Take any number of sticks from the first pile. * Take any number of sticks from the second pile. * Take the same number of sticks from both piles. (Both piles need to have at least that number of sticks.)   You win by taking the last stick. A sample game could proceed as:   * Start with the pair (6, 2). * The left player takes three sticks from the first pile. Now the game is: (3, 2) * The right player takes one stick from both piles. Now the game is: (2, 1). * The left player takes one stick from the second pile. Now the game is: (2, 0). * The right player wins by taking both sticks from the first pile. There are no more sticks, so left loses.   **Part 33, 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 34, 0 points:**These files are code that handles playing the games. CombinatorialGame is the abstract super class for all the different games. Player is the abstract super class for players of games. The Referee is a concrete (non-abstract) class that plays games between two players. It presents a player with the current game, and waits for that player to respond with a move. Once that move is returned, the ref checks that it's a legal move and then hands that new game to the other player. It waits for this player to respond, checks that the response is legal, and continues on. If the response is ever illegal, the referee forfeits that game for that player. PositionFactory is the abstract super class for objects that create game states. Referees use these factories to generate the starting position in a match. RandomPlayer is a concrete player that chooses a random legal move. I've included this as a class file instead of giving you the source code, mostly because I don't want you to use randomness when choosing a move yourself. Download all these files and put them in the same folder as your Pair.java. Feel free to look through any of the code and ask me any questions about things in them!  **Part 35, 0 points:**I coded up rules for Wythoff's Nim in [WythoffsNim.java](https://turing.plymouth.edu/~kgb1013/DB/2381/code/PairProject/WythoffsNim.java). Download this to that same folder and compile it with your code to make sure everything works.  **Part 36, 0 points:**Create an entirely new Java file where we'll test our code. (I called mine TestingMonkey.java.) Since this is only used for testing purposes, I forego the Javadoc. Add the class signature and then add a main method:  public static void main(String[] args) {  int maxPileSize = 8;  PositionFactory<WythoffsNim> factory = new WythoffsNim.PositionBuilder(maxPileSize);  Player<WythoffsNim> random = new RandomPlayer<WythoffsNim>();  Referee<WythoffsNim> ref = new Referee<WythoffsNim>(random, random, factory);  ref.call();  }  Compile and run this class. You should see a game of Wythoff's Nim being played by two players. The WythoffsNim class relies heavily on the Pair class, so if there are any exceptions, then there's probably a problem with your code. If you'd like to run a bunch of games, you can add the line:  ref.gauntlet(10000);  With this the referee will run 10,000 games and keep track of each winner. It also keeps track of the number of forfeits for each player, caused by making illegal moves. The random players I defined shouldn't forfeit any games, so if those aren't zero at this point, there's probably a problem with your code.  **Part 37, 0 points:**Now it's time to create your own code to play the game. If you take a close look at Player.java, you'll notice there are two methods you can override: getMove and toString. Create a new class file (this time using Javadoc) called WythoffsNimPlayer.java to hold the code for your player. You want to extend the player class, so your signature should look like:  public class WythoffsNimPlayer extends Player<WythoffsNim> {  If you try compiling at this point, it won't work because your class doesn't override the necessary abstract methods in Player.  **Part 38, 0 points:**Add a constructor which takes no parameters:  public WythoffsNimPlayer() {  //no state necessary  }  **Part 39, 0 points:**Override the toString method by just returning a cool name for your player. You should try to choose a name that no one else in the class will choose. For example:  public String toString() {  return "SuperRobotMonkeyTeamHyperForce";  }  **Part 40, 0 points:**Now let's implement the getMove method. This is the method a referee will call when it wants you to make a move. The first thing I like to do is throw an exception when there are no legal moves. In Wythoff's Nim, that happens when there are no sticks left in the piles. Here's my code to do that:  public WythoffsNim getMove(WythoffsNim position, int playerId) {  Pair<Integer, Integer> piles = position.getPiles();  if (piles.getFirst().equals(0) && piles.getSecond().equals(0)) {  throw new IllegalArgumentException("Given game has empty piles!");  }  //code to choose a move will go here.  }  Afterwards, I like to first make the most basic legal move possible. Let's make the most basic move by taking one stick from the first pile:  public WythoffsNim getMove(WythoffsNim position, int playerId) {  Pair<Integer, Integer> piles = position.getPiles();  if (piles.getFirst().equals(0) && piles.getSecond().equals(0)) {  throw new IllegalArgumentException("Given game has empty piles!");  }  Integer firstPile = piles.getFirst();  Pair<Integer, Integer> newPiles = new Pair<Integer, Integer>(firstPile - 1, piles.getSecond());  return new WythoffsNim(newPiles);  }  WythoffsNim's getPiles method gets us the data structure that holds the state of the game. From there, I call the getter methods and use that information to determine which move to make. In order to create the new move, I create a new pair and wrap that into a game position using the constructor call in the last line. Compile this, and modify the main method in your testing class to create a new player:  public static void main(String[] args) {  ...  Player<WythoffsNim> random = new RandomPlayer<WythoffsNim>();  Player<WythoffsNim> me = new WythoffsNimPlayer();  Referee<WythoffsNim> ref = new Referee<WythoffsNim>(random, me, factory);  ref.call();  ref.gauntlet(10000);  }  Compile and run this to see how well this player does. You should notice that it forfeits many games! That is because in some games, the first pile already has zero sticks; taking one away leaves a pile with negative sticks, which is an illegal move. We can fix this problem by first checking that the first pile has more than one stick. Modify getMove as follows:  public WythoffsNim getMove(WythoffsNim position, int playerId) {  ...  }  Integer firstPile = piles.getFirst();  //declare the pair here so we can use it outside of the following conditional  Pair<Integer, Integer> newPiles;  if (firstPile > 0) {  //take one from the first pile  newPiles = new Pair<Integer, Integer>(firstPile - 1, piles.getSecond());  } else {  //take one from the second pile  Integer secondPile = piles.getSecond();  newPiles = new Pair<Integer, Integer>(firstPile, secondPile - 1);  }  return new WythoffsNim(newPiles);  }  Compile and run your testing code again. This time your player shouldn't forfeit at all.  **Part 41, 0 points:**Spend some time playing Wythoff's Nim on paper by yourself and with other people. See if you can work out some ideas for good strategies. Implement those strategies to improve your player. (Feel free to share your thoughts with other people, but keep your source code to yourself.) Make sure:   * You always make legal moves and don't forfeit any games. * You don't use any randomness. * You don't call the getOptions method of the game object.   **Part 42, 5 points:**Improve your player so it consistently beats the random player over 50% of the time. (I recommend increasing the maximum pile size, as I'll be testing it with numbers much higher than 8.)  **Part 43, 5 points:**I've written a medium-level player: [WythoffsNimMediocrePlayer.class](https://turing.plymouth.edu/~kgb1013/DB/2381/code/PairProject/WythoffsNimMediocrePlayer.class). Download the file and play against this harder player. (The constructor takes no arguments. How do you think you would create an instance of this player? See me if you can't get this to happen.) Tweak your code to regularly beat the mediocre player 30% of the time.  **Part 44, 20 points (Bonus):**You can earn bonus points by writing an even better player. I've written a perfect Wythoff's Nim player: [WythoffsNimPerfectPlayer.class](https://turing.plymouth.edu/~kgb1013/DB/2381/code/PairProject/WythoffsNimPerfectPlayer.class). (This constructor also takes no arguments.) Keep improving your code and regularly defeat:   * The mediocre player 45% of the time for 5 bonus points. * The perfect player 20% of the time for 10 bonus points. * The perfect player 35% of the time for 15 bonus points. * The perfect player 48% of the time for 20 bonus points. If you complete this part, you've probably written a perfect player yourself! Nice job!   **Part 45, 10 points (Bonus):**Crap! The perfect player I had up was not actually perfect. I've created an actual perfect player, WythoffsNimActualPerfectPlayer: [WythoffsNimActualPerfectPlayer.class](https://turing.plymouth.edu/~kgb1013/DB/2381/code/PairProject/WythoffsNimActualPerfectPlayer.class). You can earn even more points by regularly defeating this actual perfect player:   * 35% of the time for 5 bonus points. * 48% of the time for 10 bonus points. If you complete this part, you've probably written a perfect player yourself! 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. Name this folder *YourLastName*Project0, all in PascalCase. (For example, my folder name would be: BurkeProject0.)  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:   * Pair.java * WythoffsNimPlayer.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. |