Deliverables

Your project files should be submitted to the grading system by the due date and time specified. Note that there is also an optional Skeleton Code assignment (ungraded) which will ensure that you have classes and methods named correctly and that you have the correct return types and parameter types. This ungraded assignment will also indicate level of coverage your tests have achieved. The files you submit to skeleton code assignment may be incomplete in the sense that method bodies have at least a return statement if applicable or they may be essentially completed files. To avoid a late penalty for the project, you must submit your files to the Completed Code assignment no later than 11:59 PM on the due date. Your grade will be determined, in part, by the tests that you pass or fail in your test file and by the level of coverage attained in your source file, as well as our usual correctness tests.

Files to submit to the grading system:

- SquareAntiprism.java, SquareAntiprismTest.java
- SquareAntiprismList.java, SquareAntiprismListTest.java

Specifications – Use arrays in this project; ArrayLists are not allowed!

Overview: This project consists of four classes: (1) SquareAntiprism is a class representing a SquareAntiprism object; (2) SquareAntiprismTest class is a JUnit test class which contains one or more test methods for each method in the SquareAntiprism class; (3) SquareAntiprismList is a class representing a SquareAntiprism list object; and (4) SquareAntiprismListTest class is a JUnit test class which contains one or more test methods for each method in the SquareAntiprismList class. *Note that there is no requirement for a class with a main method in this project.*

You should create a new folder to hold the files for this project and add your files from Part 2 (SquareAntiprism.java file and SquareAntiprismTest.java). You should create a new jGRASP project for Part 3 and add SquareAntiprism.java file and SquareAntiprismTest.java to the project; you should see the two files in their respective categories – Source Files and Test Files. If SquareAntiprismTest.java appears in source File category, you should right-click on the file and select "Mark As Test" from the right-click menu. You will then be able to run the test file by clicking the JUnit run button on the Open Projects toolbar. After SquareAntiprismList.java and SquareAntiprismListTest.java are created as specified below, these should be added to your jGRASP project for Part 3 as well.

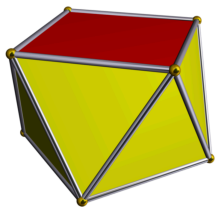
If you have successfully completed SquareAntiprism.java and SquareAntiprismTest.java in Part 2, you should go directly to SquareAntiprismList.java on page 5.

- **SquareAntiprism.java** (*new items for this class in Part 2 are underlined*)

  Requirements: Create a SquareAntiprism class that stores the label and edge (edge must be non-negative, >= 0). The SquareAntiprism class also includes methods to set and get each of these fields, as well as methods to calculate the height, surface area, and volume of a SquareAntiprism object, and a method to provide a String value that describes a SquareAntiprism object. The SquareAntiprism class includes a one static field (or class variable) to track the number of SquareAntiprism objects that have been created, as well appropriate static methods to access and

reset this field. And finally, this class provides a method that JUnit will use to test SquareAntiprism objects for equality as well as a method required by Checkstyle. In addition, SquareAntiprism must implement the Comparable interface for objects of type SquareAntiprism.

A **uniform Square Antiprism** is a Square Antiprism composed of a sequence of eight equilateral triangle sides closed by two square caps with edge length ***a***.



The variables in the formulas are:
a*: edge length*
h*: height*
A*: surface area*
V*: volume*

$$\boldsymbol{h} = \sqrt{1 - \frac{\left(\frac{1}{\cos\left(\frac{\pi}{16}\right)}\right)^2}{4}} * a$$

$$\boldsymbol{A} = 4 * ((\cos(\pi/8)/\sin(\pi/8)) + \sqrt{3}) * a^2$$

$$\mathbf{V} = 8 * \sqrt{4 * \left(\cos\left(\frac{\pi}{16}\right)\right)^2 - 1} * \sin\left(\frac{3\pi}{16}\right) * a^3/(12 * \left(\sin\left(\frac{\pi}{8}\right)\right)^2)$$

Design: The SquareAntiprism class implements the Comparable interface for objects of type SquareAntiprism and has fields, a constructor, and methods as outlined below (last method is new).

(1) **Fields:** *Instance Variables* - label of type `String` and edge of type `double`. Initialize the `String` to `""` and the `double` variable to 0 in their respective declarations. These instance variables should be private so that they are not directly accessible from outside of the SquareAntiprism class, and these should be the only instance variables (fields) in the class. *Class Variable* - count of type int should be private and static, and it should be initialized to zero.

(2) **Constructor**: Your SquareAntiprism class must contain a public constructor that accepts two parameters (see types of above) representing the label and edge. Instead of assigning the parameters directly to the fields, the respective set method for each field (described below) should be called since they are checking the validity of the parameter. For example, instead of using the statement `label = labelIn;` use the statement `setLabel(labelIn);` The constructor should increment the class variable count each time a SquareAntiprism is constructed.

Below are examples of how the constructor could be used to create SquareAntiprism objects. Note that although String and numeric literals are used for the actual parameters (or arguments) in these examples, variables of the required type could have been used instead of the literals.

```
SquareAntiprism ex1 = new SquareAntiprism("Small Example", 1.25);

SquareAntiprism ex2 = new SquareAntiprism(" Medium Example ", 10.4);

SquareAntiprism ex3 = new SquareAntiprism("Large Example", 32.46);
```

(3) **Methods**: Usually a class provides methods to access and modify each of its instance variables (known as get and set methods) along with any other required methods. The methods for SquareAntiprism, which should each be public, are described below. See the formulas in the figure above and the Code and Test section below for information on constructing these methods.

- `getLabel`: Accepts no parameters and returns a `String` representing the label field.
- `setLabel`: Takes a `String` parameter and returns a `boolean`. If the `String` parameter is not `null`, then the "trimmed" `String` is set to the label field and the method returns `true`. Otherwise, the method returns `false` and the label is not set.
- `getEdge`: Accepts no parameters and returns a `double` representing the edge field.
- `setEdge`: Takes a `double` parameter and returns a `boolean`. If the `double` parameter is non-negative, then the parameter is set to the edge field and the method returns `true`. Otherwise, the method returns `false` and the edge field is not set.
- `height`: Accepts no parameters and returns a `double` representing the height of the SquareAntiprism.
- `surfaceArea`: Accepts no parameters and returns the `double` value for the surface area of the SquareAntiprism.
- `volume`: Accepts no parameters and returns the double value for the volume of the SquareAntiprism.
- `toString`: Returns a `String` containing the information about the SquareAntiprism object formatted as shown below, including decimal formatting ("#,##0.0##") for the `double` values. Newline and tab escape sequences should be used to achieve the proper layout within the String but it should not begin or end with a newline. In addition to the field values (or corresponding "get" methods), the following methods should be used to compute appropriate values in the `toString` method: `height()`, `surfaceArea()`, and `volume()`. Each line should have no trailing spaces (e.g., there should be no spaces before a newline (\n) character). The `toString` value for `ex1`, `ex2`, and `ex3` respectively are shown below (the blank lines are not part of the `toString` values).

```
SquareAntiprism "Small Example" with edge of 1.25 units has:
   height = 1.075 units
   surface area = 25.914 square units
   volume = 8.336 cubic units

SquareAntiprism "Medium Example" with edge of 10.4 units has:
   height = 8.947 units
   surface area = 1,793.84 square units
   volume = 4,800.871 cubic units

SquareAntiprism "Large Example" with edge of 32.46 units has:
   height = 27.925 units
```

```
surface area = 17,474.872 square units
volume = 145,970.655 cubic units
```

- `getCount`: A static method that accepts no parameters and returns an int representing the static count field.
- `resetCount`: A static method that returns nothing, accepts no parameters, and sets the static count field to zero.
- `equals`: An instance method that accepts a parameter of type Object and returns false if the Object is a not a SquareAntiprism; otherwise, when cast to a SquareAntiprism, if it has the same field values (ignoring case in the label field) as the SquareAntiprism upon which the method was called, it returns true. Otherwise, it returns false. Note that this equals method with parameter type Object will be called by the JUnit Assert.assertEquals method when two SquareAntiprism objects are checked for equality.

  Below is a version you are free to use.

```
public boolean equals(Object obj) {

   if (!(obj instanceof SquareAntiprism)) {
      return false;
   }
   else {
      SquareAntiprism d = (SquareAntiprism) obj;
      return (label.equalsIgnoreCase(d.getLabel())
               && (Math.abs(edge - d.getEdge()) < .000001));
   }
}
```

- `hashCode()`: Accepts no parameters and returns zero of type int. This method is required by Checkstyle if the `equals` method above is implemented.
- `compareTo`: Accepts a parameter of type SquareAntiprism and returns an int as follows: a negative value if `this.volume()` is less than the parameter's volume; a positive value if `this.volume()` is greater than the parameter's volume; zero if the two volumes are essentially equal. *For a hint, see the activity for this module*.

Code and Test: As you implement the methods in your SquareAntiprism class, you should compile it and then create test methods as described below for the SquareAntiprismTest class.

- **SquareAntiprismTest.java**

Requirements: Create a SquareAntiprismTest class that contains a set of *test* methods to test each of the methods in SquareAntiprism. The goal for Part 2 is method, statement, and condition coverage.

Design: Typically, in each test method, you will need to create an instance of SquareAntiprism, call the method you are testing, and then make an assertion about the expected result and the actual result (note that the actual result is commonly the result of invoking the method unless it has a void return type). You can think of a test method as simply formalizing or codifying what

you could be doing in jGRASP interactions to make sure a method is working correctly. That is, the sequence of statements that you would enter in interactions to test a method should be entered into a single test method. You should have sufficient test methods so that each method, statement, and condition in SquareAntiprism are covered. Collectively, these test methods are a set of test cases that can be invoked with a single click to test all the methods in your SquareAntiprism class.

Code and Test: A good strategy would be to begin by writing test methods for those methods in SquareAntiprism that you "know" are correct. By doing this, you will be able to concentrate on the getting the test methods correct. That is, if the test method *fails*, it is most likely due to a defect in the test method itself rather the SquareAntiprism method being testing. As you become more familiar with the process of writing test methods, you will be better prepared to write the test methods as new methods are developed. Be sure to call the SquareAntiprism `toString` method in one of your test methods and assert something about the return value. If you do not want to use assertEquals, which would require the return value match the expected value exactly, you could use assertTrue and check that the return value contains the expected value. For example, for SquareAntiprism example3:

```
Assert.assertTrue(example3.toString().contains("\"Large  Example\""));
```

Also, remember that you can set a breakpoint in a JUnit test method and run the test file in Debug mode. Then, when you have an instance in the Debug tab, you can unfold it to see its values or you can open a canvas window and drag items from the Debug tab onto the canvas. You can also step-in to the method being called by the test method and then single-step through it, looking for the error.

- **SquareAntiprismList.java** (new for Part 3) – Consider implementing this file in parallel with its test file, SquareAntiprismListTest.java, which is described after this class.

  Requirements: Create a SquareAntiprismList class that stores the name of the list and an array of SquareAntiprism objects. It also includes methods that return the name of the list, number of SquareAntiprism objects in the SquareAntiprismList, total surface area, total volume, average surface area, and average volume for all SquareAntiprism objects in the SquareAntiprismList. The toString method returns summary information about the list (see below).

  Design: The SquareAntiprismList class has three fields, a constructor, and methods as outlined below.

  (1) Fields (or instance variables): (1) a String representing the name of the list, (2) an array of SquareAntiprism objects, and (3) an `int` representing the number of SquareAntiprism objects in the array, which may be less than the length of the array of SquareAntiprism objects. These instance variables should be private so that they are not directly accessible from outside of the SquareAntiprismList class. These should be the only fields (or instance variables) in this class, and they should be initialized in the constructor described below.

  (2) Constructor: Your SquareAntiprismList class must contain a constructor that accepts three parameters: (1) a parameter of type String representing the name of the list, (2) a parameter of

type `SquareAntiprism[]`, representing the list of SquareAntiprism objects, and (3) a parameter of type `int` representing the number of SquareAntiprism objects in the array. These parameters should be used to assign the fields described above (i.e., the instance variables).

(3) Methods: The methods for SquareAntiprismList are described below.

- `getName`: Returns a String representing the name of the list.
- `numberOfSquareAntiprisms`: Returns an int (the value of the third field in the SquareAntiprismList object) representing the number of SquareAntiprism objects in the SquareAntiprismList.
- `totalSurfaceArea`: Returns a double representing the total surface areas for all SquareAntiprism objects in the list. If there are zero SquareAntiprism objects in the list, zero should be returned.
- `totalVolume`: Returns a double representing the total volumes for all SquareAntiprism objects in the list. If there are zero SquareAntiprism objects in the list, zero should be returned.
- `averageSurfaceArea`: Returns a double representing the average surface area for all SquareAntiprism objects in the list. If there are zero SquareAntiprism objects in the list, zero should be returned.
- `averageVolume`: Returns a double representing the average volume for all SquareAntiprism objects in the list. If there are zero SquareAntiprism objects in the list, zero should be returned.
- `toString`: Returns a String (does not begin with \n) containing the name of the list (which can change depending on the name of the list passed as a parameter to the constructor) followed by various summary items: number of SquareAntiprisms, total surface area, total volume, average surface area, and average volume. Use "#,##0.0##" as the pattern to format the double values. Below is an example of the formatted String returned by the `toString` method, where the name of the list (name field) is `SquareAntiprism Test List` and the array of SquareAntiprism objects contains the three examples described above (top of page 3).

```
----- Summary for SquareAntiprism Test List -----
Number of SquareAntiprisms: 3
Total Surface Area: 19,294.626 square units
Total Volume: 150,779.862 cubic units
Average Surface Area: 6,431.542 square units
Average Volume: 50,259.954 cubic units
```

- `getList`: Returns the array of SquareAntiprism objects (the second field above).
- `addSquareAntiprism`: Returns nothing but takes two parameters (label and edge), creates a new SquareAntiprism object, and adds it to the SquareAntiprismList object in the next available location in the SquareAntiprism array. Be sure to increment the int field containing the number of SquareAntiprism objects in the SquareAntiprismList object.
- `findSquareAntiprism`: Takes a label of a SquareAntiprism as the String parameter and returns the corresponding SquareAntiprism object if found in the SquareAntiprismList object; otherwise returns null. Case should be ignored when attempting to match the label.

- `deleteSquareAntiprism`: Takes a String as a parameter that represents the label of the SquareAntiprism and returns the SquareAntiprism if it is found in the SquareAntiprismList object and deleted; otherwise returns null. Case should be ignored when attempting to match the label. When an element is deleted from an array, elements to the right of the deleted element must be shifted to the left. After shifting the items to the left, the last SquareAntiprism element in the array should be set to null. Finally, the number of elements field must be decremented.
- editSquareAntiprism: Takes two parameters (label and edge), uses the label to find the corresponding the SquareAntiprism object in the list. If found, sets the edge to the valuebpassed in as a parameter, and returns true. If not found, returns false. (*Note that the label should not be changed by this method*.)
- `findSquareAntiprismWithLargestVolume`: Returns the SquareAntiprism with the largest volume; if the list contains no SquareAntiprism objects, returns null.

Code and Test: Some of the methods above require that you use a loop to go through the objects in the array. You should implement the class below in parallel with this one to facilitate testing. That is, after implementing one to the methods above, you can implement the corresponding test method in the test file described below.

- **SquareAntiprismListTest.java** (new for Part 3) – Consider implementing this file in parallel with its source file, SquareAntiprismList.java, which is described above this class.

  Requirements: Create a SquareAntiprismListTest class that contains a set of *test* methods to test each of the methods in SquareAntiprismList.

  Design: Typically, in each test method, you will need to create an instance of SquareAntiprismList, call the method you are testing, and then make an assertion about the expected result and the actual result (note that the actual result is usually the result of invoking the method unless it has a void return type). You can think of a test method as simply formalizing or codifying what you have been doing in interactions to make sure a method is working correctly. That is, the sequence of statements that you would enter in interactions to test a method should be entered into a single test method. You should have at least one test method for each method in SquareAntiprismList. However, if a method contains conditional statements (e.g., an *if* statement) that results in more than one distinct outcome, you need a test method for each outcome. For example, if the method returns boolean, you should have one test method where the expected return value is false and another test method that expects the return value to be true. Also, each condition in boolean expression must be exercised true and false. Collectively, these test methods are a set of test cases that can be invoked with a single click to test all the methods in your SquareAntiprismList class.

  Code and Test: A good strategy would be to begin by writing test methods for those methods in SquareAntiprismList that you "know" are correct. By doing this, you will be able to concentrate on the getting the test methods correct. That is, if the test method *fails*, it is most likely due to a defect in the test method itself rather the SquareAntiprismList method being testing. As you become more familiar with the process of writing test methods, you will be better prepared to

write the test methods for the new methods in SquareAntiprismList. Be sure to call the SquareAntiprismList toString method in one of your test cases so that the grading system will consider the toString method to be "covered" in its coverage analysis. Remember that when a test method fails, you can set a breakpoint in a JUnit test method and run the test file in Debug mode. Then, when you have an instance in the Debug tab, you can unfold it to see its values or you can open a canvas window and drag items from the Debug tab onto the canvas. You can also step-in to the method being called by the test method and then single-step through it, looking for the error.

Finally, when comparing two arrays for equality in JUnit, be sure to use Assert.assertArrayEquals rather than Assert.assertEquals. Assert.assertArrayEquals will return true only if the two arrays are the same length and the elements are equal based on an element-by-element comparison using the equals method.

The Grading System

When you submit your files (SquareAntiprism.java, SquareAntiprismTest.java, SquareAntiprismList.java, and SquareAntiprismListTest.java), the grading system will use the results of your test methods and their level of coverage of your source files as well as the results of our reference correctness tests to determine your grade. In this project, your test files should provide method, statement, and condition coverage. Each condition in your source file must be exercised both true and false. See below for a description of how to test a boolean expression with multiple conditions.

Note For Testing the `equals` Method in SquareAntiprism

Perhaps the most complicated method to test is the `equals` method in SquareAntiprism. This method has two conditions in the boolean expression that are &&'d. Since Java (and most other languages) uses short-cut logic, if the first condition in an && is false, the &&'d expression itself is false (without considering the second condition). This means that to test the second condition, the first condition must be true. To have condition coverage for the equals method, you need the three test cases where the two conditions evaluate to the following, where T is true, F is false, and X is don't care (could be true or false):

FX - returns false
TF - returns false
TT - returns false