Due: Skeleton Code (ungraded - checks class, method names, etc.)
7A Completed Code – **Thursday, March 31, 2016 by 11:59 p.m.** (graded with your test file)
7B Completed Code – **Monday, April 4, 2016 by 11:59 p.m.** (graded with Web-CAT tests)

Deliverables

Your project files should be submitted to Web-CAT by the due date and time specified above (see the Lab Guidelines for information on submitting project files). Note that there is also an optional Skeleton Code assignment this week which will indicate level of coverage your tests have achieved. You must submit your completed code files to Web-CAT before 11:59 PM on the due date for the completed code to avoid a late penalty for the project. You may submit your completed code up to 24 hours after the due date, but there is a late penalty of 15 points. No projects will be accepted after the one day grace period. If you are unable to submit via Web-CAT, you should e-mail your project Java files in a zip file to your lab instructor before the deadline. The grade for the **7A Completed Code** submission (HexagonalPrism.java, HexagonalPrismTest.java, HexagonalPrismList2.java, and HexagonalPrismList2Test.java) will be determined by the tests that you pass or fail in your test files and by the level of coverage attained in your source files. The **7B Completed Code** will be tested against the test methods in your test files as well as usual correctness tests in Web-CAT. The 7B Completed Code assignment will not be posted until after 7A Completed Code assignment is closed at 11:59 p.m., Friday, October 30, 2015 (i.e., after the late submission day).

Files to submit to Web-CAT:

HexagonalPrism.java, HexagonalPrismTest.java
HexagonalPrismList2.java, HexagonalPrismList2Test.java

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

Overview: This project will create four classes: (1) HexagonalPrism is a class representing a HexagonalPrism object; (2) HexagonalPrismTest class is a JUnit test class which contains one or more test methods for each method in the HexagonalPrism class; (3) HexagonalPrismList2 is a class representing a HexagonalPrism list object; and (4) HexagonalPrismList2Test class is a JUnit test class which contains one or more test methods for each method in the HexagonalPrismList2 class. Note that there is no requirement for a class with a main method in this project.

Since you'll be modifying classes from the previous project, I strongly recommend that you create a new folder for this project with a copy of your HexagonalPrism class and HexagonalPrismList2 class from the previous project.

You should create a jGRASP project and add your HexagonalPrism class and HexagonalPrismList2 class. With this project is open, your test files will be automatically added to the project when they are created. You will be able to run all test files by clicking the JUnit run button on the Open Projects toolbar.

New requirements and design specifications are underlined in the descriptions below to help you identify them.

- **HexagonalPrism.java** (a modification of the **HexagonalPrism** class in the previous project; new requirements are underlined below)

  Requirements: Create a HexagonalPrism class that stores the label, side, and height. The HexagonalPrism class also includes methods to set and get each of these fields, as well as methods to calculate the base perimeter, base area, surface area, and volume of a HexagonalPrism object, and a method to provide a String value of a HexagonalPrism object (i.e., a class instance).

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

  (1) **Fields** (instance variables): label of type String, side of type double, and height of type double. These instance variables should be private so that they are not directly accessible from outside of the HexagonalPrism class, and these should be the only instance variables in the class. The class variable count should be private and static of type int, and it should be initialized to zero.

  (2) **Constructor**: Your HexagonalPrism class must contain a constructor that accepts three parameters (see types of above) representing the label, side, and height. The value for label should be trimmed of leading and trailing spaces prior to setting the field (hint: use trim method from the String class). The constructor should increment the count field each time a HexagonalPrism is constructed.

  Below are examples of how the constructor could be used to create HexagonalPrism 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.

```
HexagonalPrism example1 = new HexagonalPrism("Short Example", 4.0, 6.0);
HexagonalPrism example2 = new HexagonalPrism(" Wide Example ", 22.1, 10.6);
HexagonalPrism example3 = new HexagonalPrism("Tall Example", 10, 200);
```

  (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 HexagonalPrism are described below.
  - `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 null, then the method returns false and the label is not set. Otherwise, the "trimmed" String is set to the label field and the method returns true.
  - `getSide`: Accepts no parameters and returns a double representing the side field.
  - `setSide`: Accepts a double parameter, sets side field, and returns nothing.
  - `getHeight`: Accepts no parameters and returns a double representing the height field.
  - `setHeight`: Accepts a double parameter, sets height field, and returns nothing.

- `basePerimeter`: Accepts no parameters and returns the double value for the base perimeter of the top (or bottom) hexagon calculated using side.
- `baseArea`: Accepts no parameters and returns the double value for the base area calculated using side.
- `surfaceArea`: Accepts no parameters and returns the double value for the total surface area calculated using side and height.
- `volume`: Accepts no parameters and returns the double value for the volume calculated using side, and height.
- `toString`: Returns a String containing the information about the HexagonalPrism object formatted as shown below, including decimal formatting ("#,##0.0##") for the double values. Newline escape sequences should be used to achieve the proper layout. In addition to the field values (or corresponding "get" methods), the following methods should be used to compute appropriate values in the toString method: basePerimeter(), baseArea(), surfaceArea(), and volume(). Each line should have no leading and no trailing spaces (e.g., there should be no spaces before a newline (\n) character). The results of printing the toString value of example1, example2, and example3 respectively are shown below.

```
"Short Example" is a hexagonal prism with side = 4.0 units and height = 6.0 units,
which has base perimeter = 24.0 units, base area = 41.569 square units,
surface area = 227.138 square units, and volume = 249.415 cubic units.

"Wide Example" is a hexagonal prism with side = 22.1 units and height = 10.6 units,
which has base perimeter = 132.6 units, base area = 1,268.926 square units,
surface area = 3,943.413 square units, and volume = 13,450.62 cubic units.

"Tall Example" is a hexagonal prism with side = 10.0 units and height = 200.0 units,
which has base perimeter = 60.0 units, base area = 259.808 square units,
surface area = 12,519.615 square units, and volume = 51,961.524 cubic units.
```

- `getCount`: A static method that accepts no parameters and returns an int representing the static count field.
- `resetCount`: A static method that accepts no parameters and set the static count field to zero.
- `equals`: An instance method that accepts a parameter of type Object and returns true if the Object is a HexagonalPrism and it has the same field values as the HexagonalPrism upon which the method was called. Otherwise, it returns false. Below is a version you are free to use.

```
public boolean equals(Object obj) {
   if (!(obj instanceof HexagonalPrism)) {
      return false;
   }
   else {
      HexagonalPrism c = (HexagonalPrism) obj;
      return (label.equals(c.getLabel())
               && Math.abs(side - c.getSide()) < .00001
               && Math.abs(height - c.getHeight()) < .00001);
   }
}
```

- `hashCode()`: Accepts no parameters and returns zero of type int. This method is required by Checkstyle if the `equals` method above is implemented.

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

- **HexagonalPrismTest.java**

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

Design: Typically, in each test method, you will need to create an instance of HexagonalPrism, 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 HexagonalPrism, except for associated getters and setters which can be tested in the same method. 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. Collectively, these test methods are a set of test cases that can be invoked with a single click to test all of the methods in your HexagonalPrism class.

Code and Test: Since this is the first project requiring you to write JUnit test methods, a good strategy would be to begin by writing test methods for those methods in HexagonalPrism 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 HexagonalPrism 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 HexagonalPrism. Be sure to call the HexagonalPrism toString method in one of your test cases so that Web-CAT will consider the toString method to be "covered" in its coverage analysis. 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.

- **HexagonalPrismList2.java** (a modification of the **HexagonalPrismList2** class in the previous project; new requirements are underlined below)

Requirements: Create a HexagonalPrismList2 class that stores the name of the list, an array of HexagonalPrism objects, and the number elements in the array. It also includes methods that return the name of the list, number of HexagonalPrism objects in the HexagonalPrismList2, total

surface area, total volume, total base perimeter, total base area, average surface area, and average volume for all HexagonalPrism objects in the HexagonalPrismList2. The toString method returns a String containing the name of the list followed by each HexagonalPrism in the array, and a summaryInfo method returns summary information about the list (see below).

Design: The HexagonalPrismList2 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 HexagonalPrism objects, and (3) the number of elements (HexagonalPrism objects) in the array. These are the only fields (or instance variables) that this class should have. The array field should be declared as an array of HexagonalPrism objects and initialized to a new array of HexagonalPrism objects with length 100 (i.e., the array can hold up to 100 HexagonalPrism objects..

```
private HexagonalPrism[] list = new HexagonalPrism[100];
```

(2) Constructor: Your HexagonalPrismList2 class must contain a constructor that accepts a parameter of type String representing the name of the list, a parameter of type `HexagonalPrism[]` representing the list of HexagonalPrism objects, and a parameter of type `int` representing the number of elements in the HexagonalPrism array. These parameters should be used to assign the fields described above (i.e., the instance variables).

(3) Methods: The methods for HexagonalPrismList2 are described below.

- `getName`: Returns a String representing the name of the list.
- `numberOfHexagonalPrisms`: Returns an int representing the number of HexagonalPrism objects in the HexagonalPrismList2. If there are zero HexagonalPrism objects in the list, zero should be returned.
- `totalSurfaceArea`: Returns a double representing the total surface areas for all HexagonalPrism objects in the list. If there are zero HexagonalPrism objects in the list, zero should be returned.
- `totalVolume`: Returns a double representing the total volumes for all HexagonalPrism objects in the list. If there are zero HexagonalPrism objects in the list, zero should be returned.
- `totalBasePerimeter`: Returns a double representing the total for the base perimeters for all HexagonalPrism objects in the list. If there are zero HexagonalPrism objects in the list, zero should be returned.
- `totalBaseArea`: Returns a double representing the total for the base areas for all HexagonalPrism objects in the list. If there are zero HexagonalPrism objects in the list, zero should be returned.
- `averageSurfaceArea`: Returns a double representing the average surface area for all HexagonalPrism objects in the list. If there are zero HexagonalPrism objects in the list, zero should be returned.
- `averageVolume`: Returns a double representing the average volume for all HexagonalPrism objects in the list. If there are zero HexagonalPrism objects in the list, zero should be returned.

- `toString`: Returns a String containing the name of the list followed by each HexagonalPrism in the array. In the process of creating the return result, this toString() method should include a while loop that calls the toString() method for each HexagonalPrism object in the list. Be sure to include appropriate newline escape sequences. For an example, see lines 3 through 21 in the output in Project 5 from HexagonalPrismListApp for the *hexagonal_prism_1.dat* input file. [Note that the toString result should **not** include the summary items in lines 22 through 30 of the example. These lines represent the return value of the summaryInfo method below.]
- `summaryInfo`: Returns a String containing the name of the list (which can change depending of the value read from the file) followed by various summary items: number of hexagonal prisms, total surface area, total volume, total base perimeter, total base area, average surface area, average volume. For an example, see lines 22 through 30 in the output in Project 5 from HexagonalPrismListApp for the *hexagonal_prism_1.dat* input file. The second example below shows the output from HexagonalPrismListApp for the *hexagonal_prism_0.dat* input file which contains a list name but no hexagonal prism data.
- `getList`: Returns the array of HexagonalPrism objects (the second field above).
- `readFile`: Takes a String parameter representing the file name, reads in the file, storing the list name and creating an array of HexagonalPrism objects, uses the list name, the array, and the number of elements stored in the array to create an HexagonalPrismList2 object, and then returns the HexagonalPrismList2 object. See note #1 under Important Considerations for the HexagonalPrismList2MenuApp class (last page) to see how this method should be called.
- `addHexagonalPrism`: Returns nothing but takes three parameters (label, side, and height) creates a new HexagonalPrism object and adds it to the HexagonalPrismList2 object at the next available location in the array. Be sure to increment the number of elements field
- `deleteHexagonalPrism`: Takes the a String as a parameter that represents the label of the HexagonalPrism and returns the HexagonalPrism if it is found in the HexagonalPrismList2 object and deleted; otherwise returns null. This method should use the String equalsIgnoreCase method when attempting to match a label in the HexagonalPrism object to delete. When an element is deleted from an array, elements to the right of the deleted element must be shifted to the left, and the number of elements field must be decremented.
- `findHexagonalPrism`: Takes a label of a HexagonalPrism as the String parameter and returns the corresponding HexagonalPrism object if found in the HexagonalPrismList2 object; otherwise returns null. This method should ignore case when attempting to match the label.
- `editHexagonalPrism`: Takes three parameters (label, side, and height), uses the label to find the corresponding the HexagonalPrism object. If found, sets the side and height to the values passed in as parameters, and returns true. If not found, simply returns false.
- `findHexagonalPrismWithSmallestSide`: Returns the HexagonalPrism with the smallest side; if the list contains no HexagonalPrism objects, returns null.
- `findHexagonalPrismWithLargestSide`: Returns the HexagonalPrism with the largest side; if the list contains no HexagonalPrism objects, returns null.

- `findHexagonalPrismWithSmallestHeight`: Returns the HexagonalPrism with the smallest height; if the list contains no HexagonalPrism objects, returns null.
- `findHexagonalPrismWithLargestHeight`: Returns the HexagonalPrism with the largest height; if the list contains no HexagonalPrism objects, returns null.

- **HexagonalPrismList2Test.java**

Requirements: Create a HexagonalPrismList2Test class that contains a set of *test* methods to test each of the methods in HexagonalPrismList2.

Design: Typically, in each test method, you will need to create an instance of HexagonalPrismList2, 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 HexagonalPrismList2. 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. Collectively, these test methods are a set of test cases that can be invoked with a single click to test all of the methods in your HexagonalPrismList2class.

Code and Test: Since this is the first project requiring you to write JUnit test methods, a good strategy would be to begin by writing test methods for those methods in HexagonalPrismList2 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 HexagonalPrismList2 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 HexagonalPrismList2. Be sure to call the HexagonalPrismList2 toString method in one of your test cases so that Web-CAT will consider the toString method to be "covered" in its coverage analysis. 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.

Web-CAT

HexagonalPrism.java, HexagonalPrismTest.java, HexagonalPrismList2.java, and HexagonalPrismList2Test.java must be submitted to the **7A** and **7B** Web-CAT assignments.

Assignment 7A – Web-CAT will use the results of your test methods and their level of coverage to determine your grade. No reference correctness tests will be included in Web-CAT for assignment 7A; the reference correctness tests are simply the JUnit test methods that we use to grade your program (as we have done on previous projects). When you submit to **7A,** Web-CAT will provide feedback on failures (if any) of your test methods as well as how well your test methods covered the methods in your source files. You may need to add test methods to your test files in order to increase your grade.

Assignment 7B – As with previous projects, **7B** will include the reference correctness tests which we use to test all of your methods. Web-CAT will use the results of these correctness tests as well as the results from your test classes to determine your project grade. If you have written good test methods in your test files and your source classes pass all of them, then they should also pass our reference correctness tests.