**Due**: Skeleton Code (ungraded - checks class, method names, etc.)
7A Completed Code – **Thursday, October 27, 2016 by 11:59 p.m.** (graded with your test file)
7B Completed Code – **Monday, October 31, 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 (Cone.java, ConeTest.java, ConeList2.java, and ConeList2Test.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:**

Cone.java, ConeTest.java
ConeList2.java, ConeList2Test.java

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

**Overview**: This project will create four classes: (1) Cone is a class representing a Cone object; (2) ConeTest class is a JUnit test class which contains one or more test methods for each method in the Cone class; (3) ConeList2 is a class representing a Cone list object; and (4) ConeList2Test class is a JUnit test class which contains one or more test methods for each method in the ConeList2 class. Note that there is no requirement for a class with a main method in this project.

**Since you will be modifying classes from the previous project, I strongly recommend that you create a new folder for this project with a copy of your Cone class and ConeList2 class from the previous project.**

**You should create a jGRASP project and add your Cone class and ConeList2 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.*

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

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

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

  (1) **Fields:** Instance Variables - label of type String, height of type double, and radius of type double. These instance variables should be private so that they are not directly accessible from outside of the Cone class, and these should be the only instance variables in the class. Class Variable - count of type int should be private and static, and it should be initialized to zero.

  (2) **Constructor**: Your Cone class must contain a constructor that accepts three parameters (see types of above) representing the label, height, and radius. Instead of assigning the parameters directly to the fields, the respective set method for each field (described below) should be called. For example, instead of the statement `label = labelIn;` use the statement `setLabel(labelIn);`

  The constructor should increment the class variable count each time a Cone is constructed.

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

```
Cone example1 = new Cone("Short Example", 3.0, 4.0);
Cone example2 = new Cone(" Wide Example ", 10.6, 22.1);
Cone example3 = new Cone("Tall Example", 100, 20);
```

  (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 Cone are described below. See formulas in Code and Test 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 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.
  - `getHeight`: Accepts no parameters and returns a double representing the height field.
  - `setHeight`: Accepts a double parameter and returns a boolean. If the height is greater than zero, sets height field and returns true. Otherwise, the method returns false and the height is not set.

- `getRadius`: Accepts no parameters and returns a double representing the radius field.
- `setRadius`: Accepts a double parameter and returns a boolean. If the radius is greater than zero, sets radius field and returns true. Otherwise, the method returns false and the radius is not set.
- `basePerimeter`: Accepts no parameters and returns the double value for the perimeter of the base circle of the cone calculated using radius.
- `baseArea`: Accepts no parameters and returns the double value for the base area calculated using radius.
- `slantHeight`: Accepts no parameters and returns the double value for the slant height calculated using height and radius.
- `sideArea`: Accepts no parameters and returns the double value for the side area calculated using radius and slant height.
- `surfaceArea`: Accepts no parameters and returns the double value for the total surface area calculated using the base area and side area.
- `volume`: Accepts no parameters and returns the double value for the volume calculated using height and radius.
- `toString`: Returns a String containing the information about the Cone 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(), slantHeight(), sideArea(), 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 toString value for example1, example2, and example3 respectively are shown below (the blank lines are not part of the toString values).

```
"Short Example" is a cone with height = 3.0 units and radius = 4.0 units,
which has base perimeter = 25.133 units, base area = 50.265 square units,
slant height = 5.0 units, side area = 62.832 square units,
surface area = 113.097 square units, and volume = 50.265 cubic units.

"Wide Example" is a cone with height = 10.6 units and radius = 22.1 units,
which has base perimeter = 138.858 units, base area = 1,534.385 square units,
slant height = 24.511 units, side area = 1,701.752 square units,
surface area = 3,236.137 square units, and volume = 5,421.495 cubic units.

"Tall Example" is a cone with height = 100.0 units and radius = 20.0 units,
which has base perimeter = 125.664 units, base area = 1,256.637 square units,
slant height = 101.98 units, side area = 6,407.617 square units,
surface area = 7,664.254 square units, and volume = 41,887.902 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 Cone; otherwise, when cast to a Cone, if it has the same field values as the Cone upon which the method was called. Otherwise, it returns false. Note that this equals method with parameter type Object will be called by the JUnit Assert.assertEquals method when two Cone objects are checked for equality.

  Below is a version you are free to use.

```
public boolean equals(Object obj) {
   if (!(obj instanceof Cone)) {
      return false;
   }
   else {
      Cone c = (Cone) obj;
      return (label.equalsIgnoreCase(c.getLabel())
               && Math.abs(height - c.getHeight()) < .00001
               && Math.abs(radius - c.getRadius()) < .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 Cone class, you should compile it and then create test methods as described below for the ConeTest class.

- **ConeTest.java**

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

**Design**: Typically, in each test method, you will need to create an instance of Cone, 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 Cone, 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 (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 of the methods in your Cone 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 Cone 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 Cone 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 Cone. Be sure to call the Cone 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.

**ConeList2.java** (a modification of the **ConeList2** class in the previous project; new requirements are underlined below)

**Requirements**: Create a ConeList2 class that stores the name of the list, an array of Cone objects, and the number elements in the array. It also includes methods that return the name of the list, number of Cone objects in the ConeList2, total surface area, total volume, total base perimeter, total base area, average surface area, and average volume for all Cone objects in the ConeList2. The toString method returns a String containing the name of the list followed by each Cone in the array, and a summaryInfo method returns summary information about the list (see below).

**Design**: The ConeList2 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 Cone objects, and (3) the number of elements in the array of Cone objects. These are the only fields (or instance variables) that this class should have. The array field should be declared as an array of Cone objects and initialized to a new array of Cone objects with length 100 (i.e., the array can hold up to 100 Cone objects..

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

**(2) Constructor**: Your ConeList2 class must contain a constructor that accepts a parameter of type String representing the name of the list, a parameter of type `Cone[]` representing the list of Cone objects, and a parameter of type `int` representing the number of elements in the Cone array. These parameters should be used to assign the fields described above (i.e., the instance variables).

**(3) Methods**: The methods for ConeList2 are described below.

- `getName`: Returns a String representing the name of the list.
- `numberOfCones`: Returns an int representing the number of Cone objects in the ConeList2. If there are zero Cone objects in the list, zero should be returned.
- `totalBasePerimeter`: Returns a double representing the total for the base perimeters for all Cone objects in the list. If there are zero Cone objects in the list, zero should be returned.
- `totalBaseArea`: Returns a double representing the total for the base areas for all Cone objects in the list. If there are zero Cone objects in the list, zero should be returned.

- `totalSlantHeight`: Returns a double representing the total for the slant heights for all Cone objects in the list. If there are zero Cone objects in the list, zero should be returned.
- `totalSideArea`: Returns a double representing the total for the side areas for all Cone objects in the list. If there are zero Cone objects in the list, zero should be returned.
- `totalSurfaceArea`: Returns a double representing the total surface areas for all Cone objects in the list. If there are zero Cone objects in the list, zero should be returned.
- `totalVolume`: Returns a double representing the total volumes for all Cone objects in the list. If there are zero Cone objects in the list, zero should be returned.
- `averageSurfaceArea`: Returns a double representing the average surface area for all Cone objects in the list. If there are zero Cone objects in the list, zero should be returned.
- `averageVolume`: Returns a double representing the average volume for all Cone objects in the list. If there are zero Cone objects in the list, zero should be returned.
- `toString`: Returns a String containing the name of the list followed by each Cone in the list. In the process of creating the return result, this toString() method should include a loop that calls the toString() method for each Cone object in the list. Be sure to include appropriate newline escape sequences. See example below for option 'p' after reading in the *cone_1.dat* input file in option 'r'.
- `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 cones, total base perimeter, total base area, total slant heights, total side areas, total surface area, total volume, average surface area, average volume. See example below for option 's' after reading in the *cone_1.dat* input file in option 'r'. Be sure to try option 's' after reading in the *cone_0.dat* input file in option 'r'.
- `getList`: Returns the array of Cone 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 Cone objects, uses the list name and the array to create a ConeList2 object, and then returns the ConeList2 object. See note #1 under Important Considerations for the ConeList2MenuApp class (last page) to see how this method should be called.
- `addCone`: Returns nothing but takes three parameters (label, height, and radius), creates a new Cone object, and adds it to the ConeList2 object.
- `findCone`: Takes a label of a Cone as the String parameter and returns the corresponding Cone object if found in the ConeList2 object; otherwise returns null. This method should ignore case when attempting to match the label.
- `deleteCone`: Takes a String as a parameter that represents the label of the Cone and returns the Cone if it is found in the ConeList2 object and deleted; otherwise returns null. This method should use the String equalsIgnoreCase method when attempting to match a label in the Cone 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. After shifting the items to the left, the last Cone element in the array should be set to null. Finally, the number of elements field must be decremented.
- `editCone`: Takes three parameters (label, height, and radius), uses the label to find the corresponding the Cone object. If found, sets the height and radius to the values passed in as parameters, and returns true. If not found, simply returns false.

- `findConeWithSmallestHeight`: Returns the Cone with the smallest height; if the list contains no Cone objects, returns null.
- `findConeWithLargestHeight`: Returns the Cone with the largest height; if the list contains no Cone objects, returns null.
- `findConeWithSmallestRadius`: Returns the Cone with the smallest radius; if the list contains no Cone objects, returns null.
- `findConeWithLargestRadius`: Returns the Cone with the largest radius; if the list contains no Cone objects, returns null.

- **ConeList2Test.java**

**Requirements**: Create a ConeList2Test class that contains a set of *test* methods to test each of the methods in ConeList2.

**Design**: Typically, in each test method, you will need to create an instance of ConeList2, 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 ConeList2. 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 ConeList2class.

**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 ConeList2 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 ConeList2 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 ConeList2. Be sure to call the ConeList2 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.

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 appropriate equals method.

**Web-CAT**

Cone.java, ConeTest.java, ConeList2.java, and ConeList2Test.java must be submitted to the **7A** and **7B** Web-CAT assignments. Note that data files cone_1.dat and cone_0.dat are available in Web-CAT for you to use in your test methods. If you want to use your own data files, they should have a .txt rather than .dat extension, and they should be included with submission to Web-CAT (i.e., just add the .txt data file to your jGRASP project in the Source Files category).

**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.