Deliverables

Your project files should be submitted to Web-CAT by the due date and time specified. Note that there is also an optional Skeleton Code assignment which will indicate level of coverage your tests have achieved (there is no late penalty since the skeleton code assignment is ungraded for this project). 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. In order to avoid a late penalty for the project, you must submit your completed code files to Web-CAT no later than 11:59 PM on the due date for the completed code assignment. If you are unable to submit via Web-CAT, you should e-mail your project Java files in a zip file to your TA before the deadline. The grades for the **Part A Completed Code** submission (two files) and **Part B Completed Code** (four files) 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 as well as usual correctness tests in Web-CAT.

Files to submit to Web-CAT:

Part A

- Ellipsoid.java, EllipsoidTest.java

Part B

- Ellipsoid.java, EllipsoidTest.java
- EllipsoidList2.java, EllipsoidList2Test.java

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

Overview: This project consists of four classes: (1) Ellipsoid is a class representing an Ellipsoid object; (2) EllipsoidTest class is a JUnit test class which contains one or more test methods for each method in the Ellipsoid class; (3) EllipsoidList2 is a class representing an Ellipsoid list object; and (4) EllipsoidList2Test class is a JUnit test class which contains one or more test methods for each method in the EllipsoidList2 class. Note that there is no requirement for a class with a main method in this project.

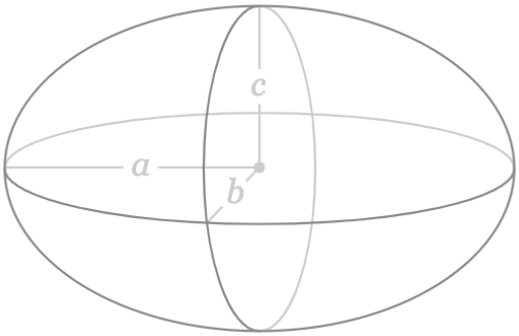
Since you will be modifying classes from the previous project, I strongly recommend that you create a new folder for this project with a copy of your Ellipsoid class and EllipsoidList2 class from the previous project.

You should create a jGRASP project and add your Ellipsoid class and EllipsoidList2 class. With this project 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.

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

  Requirements: Create an Ellipsoid class that stores the label and three axes a, b, and c. The values of the axes must be greater than zero. The Ellipsoid class also includes methods to set and get each of these fields, as well as methods to calculate the volume and surface area of the Ellipsoid object, and a method to provide a String value of an Ellipsoid object (i.e., a class instance).

An **Ellipsoid** is a 3-D object whose plane sections are ellipses defined by three axes (*a*, *b*, *c*) as depicted below. The formulas are provided to assist you in computing return values for the respective methods in the Ellipsoid class described in this project.



Formulas for volume (V) and surface area (S) are shown below.

$$V = \frac{4\pi abc}{3}$$

$$S \approx 4\pi\left(\frac{(a\,b)^{1.6}+(a\,c)^{1.6}+(b\,c)^{1.6}}{3}\right)^{1/1.6}$$

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

(1) **Fields** (three instance variables and one class variable):
Instance Variables – label of type String, and axes a, b, and c of type double. Initialize the String variable to `""` and the double variables to 0 in their respective declarations. These instance variables should be private so that they are not directly accessible from outside of the Ellipsoid class, and these should be the only instance variables (i.e., fields) in the class.
Class Variable – count of type int should be private and static, and it should be initialized to zero. This class variable is used to count the number of Ellipsoid objects created, and it should be the only class variable.

(2) **Constructor**: Your Ellipsoid class must contain a public constructor that accepts four parameters (see types of above) representing the label, a, b, and c. 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);` Below are examples of how the constructor could be used to create Ellipsoid 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.
The constructor should increment the class variable count each time an Ellipsoid is constructed.

```
Ellipsoid ex1 = new Ellipsoid ("Ex 1", 1, 2, 3);

Ellipsoid ex2 = new Ellipsoid (" Ex 2   ", 2.3, 5.5, 7.4);

Ellipsoid ex3 = new Ellipsoid ("Ex 3", 123.4, 234.5, 345.6);
```

(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 Ellipsoid, which should each be public, 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 label field is set to the "trimmed" String and the method returns true. Otherwise, the method returns false and the label field is not set.
- `getA`: Accepts no parameters and returns a double representing field *a*.
- `setA`: Accepts a double parameter and returns a boolean as follows. If the double is greater than zero, sets field *a* to the double passed in and returns true. Otherwise, the method returns false and does not set the field.
- `getB`: Accepts no parameters and returns a double representing field *b*.
- `setB`: Accepts a double parameter and returns a boolean as follows. If the double is greater than zero, sets field *b* to the double passed in and returns true. Otherwise, the method returns false and does not set the field.
- `getC`: Accepts no parameters and returns a double representing field *c*.
- `setC`: Accepts a double parameter and returns a boolean as follows. If the double is greater than zero, sets field *c* to the double passed in and returns true. Otherwise, the method returns false and does not set the field.
- `volume`: Accepts no parameters and returns the double value for the volume calculated using formula above and the values of axes fields *a*, *b*, *c*.
- `surfaceArea`: Accepts no parameters and returns the double value for the surface area calculated using formula above and the values of axes fields *a*, *b*, *c*.
- `toString`: Returns a String containing the information about the Ellipsoid 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. In addition to the field values (or corresponding "get" methods), the following methods should be used to compute appropriate values in the `toString` method: `volume()` and `surfaceArea()`. 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).

```
Ellipsoid "Ex 1" with axes a = 1.0, b = 2.0, c = 3.0 units has:
   volume = 25.1327 cubic units
   surface area = 48.9366 square units

Ellipsoid "Ex 2" with axes a = 2.3, b = 5.5, c = 7.4 units has:
   volume = 392.1127 cubic units
   surface area = 317.9245 square units

Ellipsoid "Ex 3" with axes a = 123.4, b = 234.5, c = 345.6 units has:
   volume = 41,890,963.5508 cubic units
   surface area = 674,164.7034 square units
```

New method for this project

- `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 an Ellipsoid. Otherwise, when cast to an Ellipsoid, if it has the same field values as the Ellipsoid 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 Ellipsoid objects are checked for equality.

  Below is a version you are free to use.

```
public boolean equals(Object obj) {

  if (!(obj instanceof Ellipsoid)) {
     return false;
  }
  else {
     Ellipsoid e = (Ellipsoid) obj;
     return (label.equalsIgnoreCase(e.getLabel())
        && Math.abs(a - e.getA()) < .000001
        && Math.abs(b - e.getB()) < .000001
        && Math.abs(c - e.getC()) < .000001);
  }
```

- `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 Ellipsoid class, you should compile it and then create test methods as described below for the EllipsoidTest class.

- **EllipsoidTest.java**

Requirements: Create an EllipsoidTest class that contains a set of *test* methods to test each of the methods in Ellipsoid.

Design: Typically, in each test method, you will need to create an instance of Ellipsoid, 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 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 Ellipsoid, 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 Ellipsoid 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 Ellipsoid 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 Ellipsoid 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 Ellipsoid. Be sure to call the Ellipsoid 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.

- **EllipsoidList2.java** (a modification of the **EllipsoidList2** class in the previous project; ***new requirements are underlined below.***)

  Requirements: Create an EllipsoidList2 class that stores the name of the list and an array of Ellipsoid objects, and the number of Ellipsoid objects in the array. It also includes methods that return the name of the list, number of Ellipsoid objects in the EllipsoidList2, total volume, total surface area, average volume, and average surface for all Ellipsoid objects in the EllipsoidList2. The toString method returns a String containing the name of the list followed by each Ellipsoid in the array, and a summaryInfo method returns summary information about the list (see below). In addition, there are four new methods for this project as noted below in the **Methods** section.

  Design: The EllipsoidList2 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 Ellipsoid objects, and (3) an `int` representing the number of Ellipsoid objects in the Ellipsoid array. These are the only fields (or instance variables) that this class should have.

  (2) Constructor: Your EllipsoidList2 class must contain a constructor that accepts a parameter of type String representing the name of the list, a parameter of type `Ellipsoid[]`, representing the list of Ellipsoid objects, and a parameter of type `int` representing the number of Ellipsoid objects in the Ellipsoid array. These parameters should be used to assign the fields described above (i.e., the instance variables).

  (3) Methods: The methods for EllipsoidList2 are described below.
  - `getName`: Returns a String representing the name of the list.

- `numberOfEllipsoids`: Returns an int representing the number of Ellipsoid objects in the list. If there are zero Ellipsoid objects in the list, zero should be returned.
- `totalVolume`: Returns a double representing the total volume for all Ellipsoid objects in the list. If there are zero Ellipsoid objects in the list, zero should be returned.
- `totalSurfaceArea`: Returns a double representing the total surface area for all Ellipsoid objects in the list. If there are zero Ellipsoid objects in the list, zero should be returned.
- `averageVolume`: Returns a double representing the average volume for all Ellipsoid objects in the list. If there are zero Ellipsoid objects in the list, zero should be returned.
- `averageSurfaceArea`: Returns a double representing the average surface area for all Ellipsoid objects in the list. If there are zero Ellipsoid objects in the list, zero should be returned.
- `toString`: Returns a String (does not begin with \n) containing the name of the list followed by each Ellipsoid in the list. In the process of creating the return result, this toString() method should include a while loop that calls the toString() method for each Ellipsoid object in the list (adding a \n before and after each). Be sure to include appropriate newline escape sequences. For an example, in the previous project see lines 3 through 16 in the output below from EllipsoidListApp for the *Ellipsoid_data_1.txt* input file. [Note that the toString result should **not** include the summary items in lines 18 through 24 of the example. These lines represent the return value of the summaryInfo method below.]
- `summaryInfo`: Returns a String (does not begin with \n) containing the name of the list (which can change depending of the value read from the file) followed by various summary items: number of Ellipsoid objects, total volume, total surface area, average volume, and average surface area. Use "#,##0.0##" as the pattern to format the double values. For an example, in the previous project see lines 18 through 24 in the output below from EllipsoidListApp for the *Ellipsoid_data_1.txt* input file. The second example below shows the output from EllipsoidListApp for the *Ellipsoid_data_0.txt* input file which contains a list name but no Ellipsoid data.
- `getList`: Returns the array of Ellipsoid objects (the second field above).
- `readFile`: Takes a String parameter representing the file name, creates an array of Ellipsoid objects with length of 100, reads in the file, storing the list name and creating Ellipsoid objects, adding them to the Ellipsoid array, then uses the list name, the array, and number of Ellipsoid objects in the array to create an EllipsoidList2 object, and finally returns the EllipsoidList2 object. See note #1 under Important Considerations for the EllipsoidList2MenuApp class (last page) to see how this method should be called.
- `addEllipsoid`: Returns nothing but takes four parameters (label, a, b, and c), creates a new Ellipsoid object, and adds it to the EllipsoidList2 object. Finally, the number of Ellipsoid objects field must be incremented.
- `findEllipsoid`: Takes a label of an Ellipsoid as the String parameter and returns the corresponding Ellipsoid object if found in the EllipsoidList2 object; otherwise returns null. Case should be ignored when attempting to match the label.
- `deleteEllipsoid`: Takes a String as a parameter that represents the label of the Ellipsoid and returns the Ellipsoid if it is found in the EllipsoidList2 object and deleted; otherwise returns null. Case should be ignored when attempting to match the label; consider calling/using `findEllipsoid` in this method. 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 Ellipsoid element in the array should be set to null. Finally, the number of Ellipsoid objects field must be decremented.

- editEllipsoid: Takes four parameters (label, a, b, and c), uses the label to find the corresponding the Ellipsoid object. If found, sets the a, b, and c to the values passed in as parameters, and returns the Ellipsoid object. If not found, returns null. ***This method should not change the label***.

New methods for this project

- `findEllipsoidWithSmallestVolume`: Returns the Ellipsoid with the smallest volume; if the list contains no Ellipsoid objects, returns null.
- `findEllipsoidWithLargestVolume`: Returns the Ellipsoid with the largest volume; if the list contains no Ellipsoid objects, returns null.
- `findEllipsoidWithSmallestSurfaceArea`: Returns the Ellipsoid with the smallest surface area; if the list contains no Ellipsoid objects, returns null.
- `findEllipsoidWithLargestSurfaceArea`: Returns the Ellipsoid with the largest surface area; if the list contains no Ellipsoid objects, returns null.

Code and Test: Remember to import java.util.Scanner, java.io.File, java.io.IOException. These classes will be needed in the readFile method which will require a throws clause for IOException. Some of the methods above require that you use a loop to go through the objects in the array. You may want to 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.

- **EllipsoidList2Test.java**

Requirements: Create an EllipsoidList2Test class that contains a set of *test* methods to test each of the methods in EllipsoidList2.

Design: Typically, in each test method, you will need to create an instance of EllipsoidList2, 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 EllipsoidList2. 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 EllipsoidList2 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 EllipsoidList2 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 EllipsoidList2 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 EllipsoidList2. Be sure to call the EllipsoidList2 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.

Important: 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

Assignment Part A – submit: Ellipsoid.java, EllipsoidTest.java

Assignment Part B – submit: Ellipsoid.java, EllipsoidTest.java, EllipsoidList2.java, and EllipsoidList2Test.java.

Note that data files *Ellipsoid_data_1.txt* and *Ellipsoid_data_0.txt* 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 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).

Web-CAT 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.