

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

- Icosahedron.java, IcosahedronTest.java

Part B

- IcosahedronList2.java, IcosahedronList2Test.java

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

Overview: This project consists of four classes: (1) Icosahedron is a class representing an Icosahedron object; (2) IcosahedronTest class is a JUnit test class which contains one or more test methods for each method in the Icosahedron class; (3) IcosahedronList2 is a class representing an Icosahedron list object; and (4) IcosahedronList2Test class is a JUnit test class which contains one or more test methods for each method in the IcosahedronList2 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 Icosahedron class and IcosahedronList2 class from the previous project.

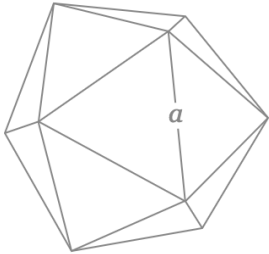
You should create a jGRASP project and add your Icosahedron class and IcosahedronList2 class. With this project is open, your test files will be automatically added to the project when they are created. You will be able to run all test files by clicking the JUnit run button on the Open Projects toolbar.

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

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

Requirements: Create an Icosahedron class that stores the label, color, and edge (i.e., length of an edge, which must be greater than zero). The Icosahedron class also includes methods to set and get each of these fields, as well as methods to calculate the surface area, volume, and surface to volume ratio of an Icosahedron object, and a method to provide a String value of an Icosahedron object (i.e., a class instance).

An **Icosahedron** has 20 equilateral triangle faces, 12 vertices, and 30 edges as depicted below. The formulas are provided to assist you in computing return values for the respective methods in the Icosahedron class described in this project.

	Surface Area (A)	$A = 5\sqrt{3}a^2$ $V = \frac{5(3+\sqrt{5})}{12}a^3$
	Volume (V)	
	Edge length (a)	
	Surface/Volume ratio (A/V)	

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

- (1) **Fields** (instance variables): label of type String, color of type String, and edge of type double. Initialize the Strings to "" and the double to 0 in their respective declarations. These instance variables should be private so that they are not directly accessible from outside of the Icosahedron 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 Icosahedron class must contain a public constructor that accepts three parameters (see types of above) representing the label, color, and edge. 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 Icosahedron 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 Icosahedron is constructed.

```
Icosahedron example1 = new Icosahedron("Small", "blue", 0.01);
```

```
Icosahedron example2 = new Icosahedron("    Medium    ", "orange", 12.3);  
Icosahedron example3 = new Icosahedron("Large", "    white    ", 123.4);
```

(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 Icosahedron, 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 “trimmed” String is set to the label field and the method returns true. Otherwise, the method returns false and the label is not set.
- `getColor`: Accepts no parameters and returns a String representing the color field.
- `setColor`: Takes a String parameter and returns a boolean. If the string parameter is not null, then the “trimmed” String is set to the color 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`: Accepts a double parameter and returns a boolean as follows. If the edge is greater than zero, sets the edge field to the double passed in and returns true. Otherwise, the method returns false and the edge is not set.
- `surfaceArea`: Accepts no parameters and returns the double value for the total surface area calculated using the value for edge.
- `volume`: Accepts no parameters and returns the double value for the volume calculated using the value for edge.
- `surfaceToVolumeRatio`: Accepts no parameters and returns the double value calculated by dividing the total surface area by the volume.
- `toString`: Returns a String containing the information about the Icosahedron 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: `surfaceArea()`, `volume()`, and `surfaceToVolumeRatio()`. Each line should have 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).

```
Icosahedron "Small" is "blue" with 30 edges of length 0.01 units.  
    surface area = 0.000866 square units  
    volume = 0.000002 cubic units  
    surface/volume ratio = 396.950723
```

```
Icosahedron "Medium" is "orange" with 30 edges of length 12.3 units.  
    surface area = 1,310.209833 square units  
    volume = 4,059.844212 cubic units  
    surface/volume ratio = 0.322724
```

```
Icosahedron "Large" is "white" with 30 edges of length 123.4 units.  
    surface area = 131,874.537977 square units
```

```
volume = 4,099,581.395236 cubic units
surface/volume ratio = 0.032168
```

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 not an Icosahedron; otherwise, when cast to an Icosahedron, if it has the same field values as the Icosahedron 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 Icosahedron objects are checked for equality.

Below is a version you are free to use.

```
public boolean equals(Object obj) {
    if (!(obj instanceof Icosahedron)) {
        return false;
    }
    else {
        Icosahedron d = (Icosahedron) obj;
        return (label.equalsIgnoreCase(d.getLabel())
            && color.equalsIgnoreCase(d.getColor())
            && 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.

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

- **IcosahedronTest.java**

Requirements: Create an IcosahedronTest class that contains a set of test methods to test each of the methods in Icosahedron.

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

- **IcosahedronList2.java** (a modification of the **IcosahedronList2** class in the previous project; new requirements are underlined below)

Requirements: Create an IcosahedronList2 class that stores the name of the list and an array of Icosahedron objects, and the number of Icosahedron objects in the array. It also includes methods that return the name of the list, number of Icosahedron objects in the IcosahedronList2, total surface area, total volume, average surface area, average volume, and average surface to volume ratio for all Icosahedron objects in the IcosahedronList2. The toString method returns a String containing the name of the list followed by each Icosahedron in the array, and a summaryInfo method returns summary information about the list (see below).

Design: The IcosahedronList2 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 Icosahedron objects, and (3) an `int` representing the number of Icosahedron objects in the Icosahedron array. These are the only fields (or instance variables) that this class should have.
- (2) **Constructor:** Your IcosahedronList2 class must contain a constructor that accepts a parameter of type String representing the name of the list, a parameter of type `Icosahedron[]`, representing the list of Icosahedron objects, and a parameter of type `int` representing the number of Icosahedron objects in the Icosahedron array. These parameters should be used to assign the fields described above (i.e., the instance variables).

(3) Methods: The methods for IcosahedronList2 are described below.

- `getName`: Returns a String representing the name of the list.
- `numberOfIcosahedrons`: Returns an int representing the number of Icosahedron objects in the IcosahedronList2. If there are zero Icosahedron objects in the list, zero should be returned.
- `totalSurfaceArea`: Returns a double representing the total surface areas for all Icosahedron objects in the list. If there are zero Icosahedron objects in the list, zero should be returned.
- `totalVolume`: Returns a double representing the total volumes for all Icosahedron objects in the list. If there are zero Icosahedron objects in the list, zero should be returned.
- `averageSurfaceArea`: Returns a double representing the average surface area for all Icosahedron objects in the list. If there are zero Icosahedron objects in the list, zero should be returned.
- `averageVolume`: Returns a double representing the average volume for all Icosahedron objects in the list. If there are zero Icosahedron objects in the list, zero should be returned.
- `averageSurfaceToVolumeRatio`: Returns a double representing the average surface to volume ratio for all Icosahedron objects in the list. If there are zero Icosahedron 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 Icosahedron 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 Icosahedron object in the list (adding a `\n` before and after each). Be sure to include appropriate newline escape sequences. For an example, see [lines 2 through 19](#) in the output below from IcosahedronList2App for the *Icosahedron_data_1.txt* input file. [Note that the `toString` result should **not** include the return value of `summaryInfo()`.]
- `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 Icosahedrons, total surface area, total volume, average surface area, average volume, and average surface to volume ratio. Use `"#,##0.0##"` as the pattern to format the double values.
- `getList`: Returns the array of Icosahedron 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 Icosahedron objects, uses the list name, the array, and number of Icosahedron objects in the array to create an IcosahedronList2 object, and then returns the IcosahedronList2 object. See note #1 under [Important Considerations](#) for the IcosahedronList2MenuApp class (last page) to see how this method should be called.
- `addIcosahedron`: Returns nothing but takes three parameters (label, color, and edge), creates a new Icosahedron object, and adds it to the IcosahedronList2 object.
- `findIcosahedron`: Takes a label of an Icosahedron as the String parameter and returns the corresponding Icosahedron object if found in the IcosahedronList2 object; otherwise returns null. Case should be ignored when attempting to match the label.

- `deleteIcosahedron`: Takes a String as a parameter that represents the label of the Icosahedron and returns the Icosahedron if it is found in the IcosahedronList2 object and deleted; otherwise returns null. Case should be ignored when attempting to match the label; consider calling/using `findIcosahedron` 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 Icosahedron element in the array should be set to null. Finally, the number of elements field must be decremented.
- `editIcosahedron`: Takes three parameters (label, color, and edge), uses the label to find the corresponding the Icosahedron object in the list. If found, sets the color and edge to the values passed in as parameters, and returns true. If not found, returns false.

New method for this project

- `findIcosahedronWithShortestEdge` : Returns the Icosahedron with the shortest edge; if the list contains no Icosahedron objects, returns null.
- `findIcosahedronWithLongestEdge` : Returns the Icosahedron with the longest edge; if the list contains no Icosahedron objects, returns null.
- `findIcosahedronWithSmallestVolume` : Returns the Icosahedron with the smallest volume; if the list contains no Icosahedron objects, returns null.
- `findIcosahedronWithLargestVolume` : Returns the Icosahedron with the largest volume; if the list contains no Icosahedron 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.

- **IcosahedronList2Test.java**

Requirements: Create an `IcosahedronList2Test` class that contains a set of *test* methods to test each of the methods in `IcosahedronList2`.

Design: Typically, in each test method, you will need to create an instance of `IcosahedronList2`, 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 `IcosahedronList2`. 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 IcosahedronList2 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 IcosahedronList2 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 IcosahedronList2 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 IcosahedronList2. Be sure to call the IcosahedronList2 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: Icosahedron.java, IcosahedronTest.java

Assignment Part B – submit: Icosahedron.java, IcosahedronTest.java, IcosahedronList2.java, and IcosahedronList2Test.java.

Note that data files icosahedron_data_1.txt and icosahedron_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.