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 this week 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 <u>completed code</u> files to Web-CAT no later than 11:59 PM on the due date for the completed code. You may submit your <u>completed code</u> 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 late period. 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 grade for the **Part A Completed Code** submission (Dodecahedron.java, DodecahedronTest.java, DodecahedronList2.java, and DodecahedronList2Test.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 **Part B 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 <u>not</u> be posted until after 7A Completed Code assignment is closed (i.e., after the late submission day).

Files to submit to Web-CAT (all three files must be submitted together):

- Dodecahedron.java, DodecahedronTest.java
- DodecahedronList2.java, DodecahedronList2Test.java

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

Overview: This project consists of four classes: (1) Dodecahedron is a class representing a Dodecahedron object; (2) DodecahedronTest class is a JUnit test class which contains one or more test methods for each method in the Dodecahedron class; (3) DodecahedronList2 is a class representing a Dodecahedron list object; and (4) DodecahedronList2Test class is a JUnit test class which contains one or more test methods for each method in the DodecahedronList2 class. <u>Note that there is no requirement for a class with a main method in this project</u>.

<u>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 Dodecahedron class and DodecahedronList2 class from the previous project</u>.

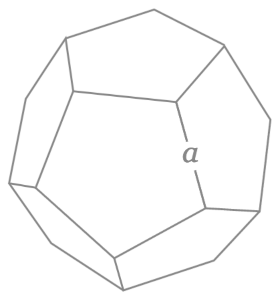
<u>You should create a jGRASP project and add your Dodecahedron class and DodecahedronList2 class. With this project is open, your test files will be automatically added to the project when they are created. You will be able to run all test files by clicking the JUnit run button on the Open Projects toolbar</u>.

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

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

  Requirements: Create a Dodecahedron class that stores the label, color, and edge (i.e., length of an edge, which must be greater than zero). The Dodecahedron 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 a Dodecahedron object, and a method to provide a String value of a Dodecahedron object (i.e., a class instance).

A dodecahedron has 12 equal pentagonal faces, 20 vertices, and 30 edges as depicted below. The formulas are provided to assist you in computing return values for the respective Dodecahedron methods described in this project.

	Surface Area (A) Volume (V) Edge length (a) Surface/Volume ratio (A/V)	$A = 3\sqrt{25+10\sqrt{5}}\,a^2$ $V = \frac{15+7\sqrt{5}}{4}a^3$

Design: The Dodecahedron 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 Dodecahedron 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 Dodecahedron 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);`

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

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

```
Dodecahedron example1 = new Dodecahedron("Small Example", "blue", 0.25);

Dodecahedron example2 = new Dodecahedron(" Medium Example ", "orange", 10.1);

Dodecahedron example3 = new Dodecahedron("Large Example", "silver  ", 200.5);
```

(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 Dodecahedron, which should each be public, are described below. See formulas in Code and Test below.

- o `getLabel`: Accepts no parameters and returns a String representing the label field.
- o `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.
- o `getColor`: Accepts no parameters and returns a String representing the color field.
- o `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.
- o `getEdge`: Accepts no parameters and returns a double representing the edge field.
- o `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.
- o `surfaceArea`: Accepts no parameters and returns the double value for the total surface area calculated using the value for edge.
- o `volume`: Accepts no parameters and returns the double value for the volume calculated using the value for edge.
- o `surfaceToVolumeRatio`: Accepts no parameters and returns the double value calculated by dividing the total surface area by the volume.
- o `toString`: Returns a String (does not begin with \n) containing the information about the Dodecahedron 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).

```
Dodecahedron "Small Example" is "blue" with 30 edges of length 0.25 units.
   surface area = 1.29 square units
   volume = 0.12 cubic units
   surface/volume ratio = 10.777

Dodecahedron "Medium Example" is "orange" with 30 edges of length 10.1 units.
   surface area = 2,106.071 square units
   volume = 7,895.319 cubic units
   surface/volume ratio = 0.267
```

```
Dodecahedron "Large Example" is "silver" with 30 edges of length 200.5 units.
   surface area = 829,963.459 square units
   volume = 61,765,889.248 cubic units
   surface/volume ratio = 0.013
```

- `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 Dodecahedron; otherwise, when cast to a Dodecahedron, if it has the same field values as the Dodecahedron 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 Dodecahedron objects are checked for equality.

  Below is a version you are free to use.

  ```
  public boolean equals(Object obj) {

     if (!(obj instanceof Dodecahedron)) {
        return false;
     }
     else {
        Dodecahedron d = (Dodecahedron) 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 Dodecahedron class, you should compile it and then create test methods as described below for the DodecahedronTest class.

- **DodecahedronTest.java**

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

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

- **DodecahedronList2.java** (a modification of the **DodecahedronList2** class in the previous project; new requirements are underlined below)

  Requirements: Create a DodecahedronList2 class that stores the name of the list and an array of Dodecahedron objects. It also includes methods that return the name of the list, number of Dodecahedron objects in the DodecahedronList2, total surface area, total volume, average surface area, average volume, and average surface to volume ratio for all Dodecahedron objects in the DodecahedronList2. The toString method returns a String containing the name of the list followed by each Dodecahedron in the array, and a summaryInfo method returns summary information about the list (see below).

  Design: The DodecahedronList2 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 Dodecahedron objects, and (3) an `int` representing the number of elements in the array of Dodecahedron objects. These are the only fields (or instance variables) that this class should have.

  (2) Constructor: Your DodecahedronList2 class must contain a constructor that accepts a parameter of type String representing the name of the list, a parameter of type `Dodecahedron[]`, representing the list of Dodecahedron objects, and a parameter of type `int` representing the number of elements in the Dodecahedron array. These parameters should be used to assign the fields described above (i.e., the instance variables).

(3) Methods: The methods for DodecahedronList2 are described below.

- `getName`: Returns a String representing the name of the list.
- `numberOfDodecahedrons`: Returns an int representing the number of Dodecahedron objects in the DodecahedronList2. If there are zero Dodecahedron objects in the list, zero should be returned.
- `totalSurfaceArea`: Returns a double representing the total surface areas for all Dodecahedron objects in the list. If there are zero Dodecahedron objects in the list, zero should be returned.
- `totalVolume`: Returns a double representing the total volumes for all Dodecahedron objects in the list. If there are zero Dodecahedron objects in the list, zero should be returned.
- `averageSurfaceArea`: Returns a double representing the average surface area for all Dodecahedron objects in the list. If there are zero Dodecahedron objects in the list, zero should be returned.
- `averageVolume`: Returns a double representing the average volume for all Dodecahedron objects in the list. If there are zero Dodecahedron objects in the list, zero should be returned.
- `averageSurfaceToVolumeRatio`: Returns a double representing the average surface to volume ratio for all Dodecahedron objects in the list. If there are zero Dodecahedron 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 Dodecahedron 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 Dodecahedron 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 DodecahedronList2App for the *Dodecahedron_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 Dodecahedrons, 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 Dodecahedron 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 Dodecahedron objects, uses the list name, the array, and number of Dodecahedron objects in the array to create a DodecahedronList2 object, and then returns the DodecahedronList2 object. See note #1 under Important Considerations for the DodecahedronList2MenuApp class (last page) to see how this method should be called.
- `addDodecahedron`: Returns nothing but takes three parameters (label, color, and edge), creates a new Dodecahedron object, and adds it to the DodecahedronList2 object.
- `findDodecahedron`: Takes a label of a Dodecahedron as the String parameter and returns the corresponding Dodecahedron object if found in the DodecahedronList2

object; otherwise returns null. Case should be ignored when attempting to match the label.

- `deleteDodecahedron`: Takes a String as a parameter that represents the label of the Dodecahedron and returns the Dodecahedron if it is found in the DodecahedronList2 object and deleted; otherwise returns null. Case should be ignored when attempting to match the label; consider calling/using `findDodecahedron` 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 Dodecahedron element in the array should be set to null. Finally, the number of elements field must be decremented.
- `editDodecahedron`: Takes three parameters (label, color, and edge), uses the label to find the corresponding the Dodecahedron 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 Project 7

- `findDodecahedronWithShortestEdge`: Returns the Dodecahedron with the shortest edge; if the list contains no Dodecahedron objects, returns null.
- `findDodecahedronWithLongestEdge`: Returns the Dodecahedron with the longest edge; if the list contains no Dodecahedron objects, returns null.
- `findDodecahedronWithSmallestVolume`: Returns the Dodecahedron with the smallest volume; if the list contains no Dodecahedron objects, returns null.
- `findDodecahedronWithLargestVolume`: Returns the Dodecahedron with the largest volume; if the list contains no Dodecahedron 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 "case" in the switch for menu described below in the DodecahedronList2MenuApp class.

- **DodecahedronList2Test.java**

  Requirements: Create a DodecahedronList2Test class that contains a set of *test* methods to test each of the methods in DodecahedronList2.

  Design: Typically, in each test method, you will need to create an instance of DodecahedronList2, 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 DodecahedronList2. 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 DodecahedronList2 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 DodecahedronList2 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 DodecahedronList2 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 DodecahedronList2. Be sure to call the DodecahedronList2 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

Dodecahedron.java, DodecahedronTest.java, DodecahedronList2.java, and DodecahedronList2Test.java must be submitted to the **Part A** and **Part B** Web-CAT assignments. Note that data files dodecahedron_data_1.txt and dodecahedron_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).

Assignment Part A – 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 Part A; 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 **Part A,** 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 Part B – As with previous projects, **Part B** 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.