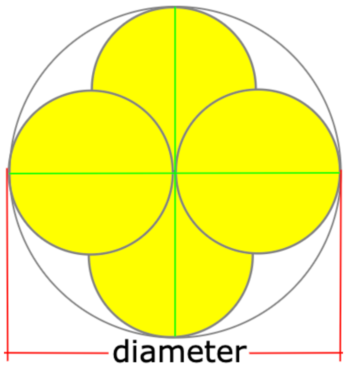
Deliverables

Your project files should be submitted to the grading system by the due date and time specified. Note that there is also an optional Skeleton Code assignment (ungraded) which will ensure that you have classes and methods named correctly and that you have the correct return types and parameter types. This ungraded assignment will also indicate level of coverage your tests have achieved. 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. To avoid a late penalty for the project, you must submit your files to the Completed Code assignment no later than 11:59 PM on the due date. Your grade will be determined, in part, by the tests that you pass or fail in your test file and by the level of coverage attained in your source file, as well as our usual correctness tests.

Files to submit to the grading system:

- Quatrefoil.java, QuatrefoilTest.java
- QuatrefoilList.java, QuatrefoilListTest.java

A **quatrefoil** is a symmetrical shape which forms the overall outline of four partially overlapping circles of the same diameter shown in yellow below. In the figure, a circumscribed circle is shown, which is used in the formulas below.



Variables in formulas:

$\boldsymbol{d}$: diameter of large circle in inches
$\boldsymbol{r}$: radius of large circle (half the diameter)
$\boldsymbol{P}$: perimeter in inches
$\boldsymbol{P_{ft}}$: perimeter in feet
$\boldsymbol{A}$: area in square inches
$\boldsymbol{A_{sqft}}$: area in square feet

$$\boldsymbol{P} = \pi d$$
$$\boldsymbol{P_{ft}} = P/12$$
$$\boldsymbol{A} = r^2 + 2(\pi r^2/4)$$
$$\boldsymbol{A_{sqft}} = A/144$$

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

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

You should create a new folder to hold the files for this project and add your files from Part 2 (Quatrefoil.java file and QuatrefoilTest.java). You should create a new jGRASP project for Part 3 and add Quatrefoil.java file and QuatrefoilTest.java to the project; you should see the two files in their respective categories – Source Files and Test Files. If QuatrefoilTest.java appears in source File category, you should right-click on the file and select "Mark As Test" from the right-click menu.

You will then be able to run the test file by clicking the JUnit run button on the Open Projects toolbar. After QuatrefoilList.java and QuatrefoilListTest.java are created as specified below, these should be added to your jGRASP project for Part 3 as well.

If you have successfully completed Quatrefoil.java and QuatrefoilTest.java in Part 2, you should go directly to QuatrefoilList.java on page 6.

- **Quatrefoil.java** (*new items for this class in Part 2 are underlined*)

  Requirements: Create a Quatrefoil class that stores the label and diameter of the large circle where diameter is positive. The Quatrefoil class also includes methods to set and get each of these fields, as well as methods to calculate the perimeter in inches, perimeter in feet, area in square inches, and area in square feet of a Quatrefoil object, and a method to provide a String value that describes a Quatrefoil object. The Quatrefoil class includes a one static field (or class variable) to track the number of Quatrefoil objects that have been created, as well appropriate static methods to access and reset this field. And finally, this class provides a method that JUnit will use to test Quatrefoil objects for equality as well as a method required by Checkstyle.
  In addition, Quatrefoil must implement the Comparable interface for objects of type Quatrefoil.

  Design: The Quatrefoil class implements the Comparable interface for objects of type Quatrefoil and has fields, a constructor, and methods as outlined below (last method is new).

  (1) **Fields:** Instance Variables - label of type `String` and diameter in inches of the large (circumscribed) circle of type `double`. Initialize the `String` to `""` and the `double` variable to 0 in their respective declarations. These instance variables should be private so that they are not directly accessible from outside of the Quatrefoil class, and these should be the only instance variables (fields) in the class.
  Class Variable - count of type int should be private and static, and it should be initialized to zero in the declaration.

  (2) **Constructor**: Your Quatrefoil class must contain a public constructor that accepts two parameters (see types of above) representing the label and diameter of the large (circumscribed) circle. Instead of assigning the parameters directly to the fields, the respective set method for each field (described below) should be called since they are checking the validity of the parameter. For example, instead of using the statement `label = labelIn;` use the statement `setLabel(labelIn);`
  After setLabel is called be sure to call setDiameter. The constructor should increment the class variable count each time a Quatrefoil is constructed.

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

```
Quatrefoil ex1 = new Quatrefoil ("Small Example", 3.5);
Quatrefoil ex2 = new Quatrefoil (" Medium Example ", 38.44);

Quatrefoil ex3 = new Quatrefoil ("Large Example", 111.4);
```

(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 Quatrefoil, which should each be public, are described below. See the formulas in the figure above and the Code and Test section below for information on constructing these methods.

- `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.
- `getDiameter`: Accepts no parameters and returns a `double` representing the diameter of the diameter field.
- `setDiameter`: Takes a `double` parameter and returns a `boolean`. If the `double` parameter is positive, then the parameter is assigned to the diameter field and the method returns `true`. Otherwise, the method returns `false` and the diameter field is not set.
- `perimeter`: Accepts no parameters and returns a `double` representing the perimeter of the Quatrefoil in inches. See formula in figure on page 1.
- `perimeterInFt`: Accepts no parameters and returns a `double` representing the perimeter of the Quatrefoil in feet. See formula in figure on page 1.
- `area`: Accepts no parameters and returns the `double` value for the area of the Quatrefoil in square inches. See formula in figure on page 1.
- `areaInSqFt`: Accepts no parameters and returns the double value for the area of the Quatrefoil in square feet. See formula in figure on page 1.
- `toString`: Returns a `String` containing the information about the Quatrefoil 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 within the String but it should not begin or end with a newline. In addition to the field values (or corresponding "get" methods), the following methods should be used to compute appropriate values in the `toString` method: `perimeter()`, `perimeterInFt()`, `area()`, and `areaInSqFt()`. 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).

```
Quatrefoil "Small Example" with diameter 3.5 inches has:
   perimeter = 10.996 inches (or 0.916 feet)
   area = 7.873 square inches (or 0.055 square feet)

Quatrefoil "Medium Example" with diameter 38.44 inches has:
   perimeter = 120.763 inches (or 10.064 feet)
   area = 949.674 square inches (or 6.595 square feet)

Quatrefoil "Large Example" with diameter 111.4 inches has:
   perimeter = 349.973 inches (or 29.164 feet)
```

```
area = 7,975.87 square inches (or 55.388 square feet)
```

- `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 Quatrefoil; otherwise, when cast to a Quatrefoil, if it has the same field values (ignoring case in the label field) as the Quatrefoil 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 Quatrefoil objects are checked for equality.

  Below is a version you are free to use.

```
public boolean equals(Object obj) {

   if (!(obj instanceof Quatrefoil)) {
      return false;
   }
   else {
      Quatrefoil qf = (Quatrefoil) obj;
      return (label.equalsIgnoreCase(qf.getLabel())
               && Math.abs(diameter - qf.getDiameter()) < .000001);
   }
}
```

- `hashCode()`: Accepts no parameters and returns zero of type int. This method is required by Checkstyle if the `equals` method above is implemented.
- `compareTo`: Accepts a parameter of type Quatrefoil and returns an int as follows: a negative value if `this.area()` is less than the parameter's area; a positive value if `this.area()` is greater than the parameter's area; zero if the two areas are essentially equal. *For a hint, see the activity for this module*.

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

- **QuatrefoilTest.java**

Requirements: Create a QuatrefoilTest class that contains a set of *test* methods to test each of the methods in Quatrefoil. The goal for Part 2 is method, statement, and condition coverage.

Design: Typically, in each test method, you will need to create an instance of Quatrefoil, 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 could be doing in jGRASP 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 Quatrefoil, except for associated getters and setters which can be tested in the same method. In addition, you should have sufficient test methods so that each method, statement, and condition in Quatrefoil are covered. Collectively, these test methods are a set of test cases that can be invoked with a single click to test all the methods in your Quatrefoil class.

Code and Test: A good strategy would be to begin by writing test methods for those methods in Quatrefoil 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 Quatrefoil 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 as new methods are developed. Be sure to call the Quatrefoil `toString` method in one of your test methods and assert something about the return value. If you do not want to use assertEquals, which would require the return value match the expected value exactly, you could use assertTrue and check that the return value contains the expected value. For example, for Quatrefoil ex3:

```
   Assert.assertTrue(ex3.toString().contains("\"Large Example\""));
```

Also, 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. You can also step-in to the method being called by the test method and then single-step through it, looking for the error.

- **QuatrefoilTest.java**

Requirements: Create a QuatrefoilTest class that contains a set of *test* methods to test each of the methods in Quatrefoil. The goal for Part 2 is method, statement, and condition coverage.

Design: Typically, in each test method, you will need to create an instance of Quatrefoil, 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 could be doing in jGRASP 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 sufficient test methods so that each method, statement, and condition in Quatrefoil are covered. Collectively, these test methods are a set of test cases that can be invoked with a single click to test all the methods in your Quatrefoil class.

Code and Test: A good strategy would be to begin by writing test methods for those methods in Quatrefoil 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 Quatrefoil 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 as new methods are developed. Be sure to call the Quatrefoil `toString` method in one of your test methods and assert something about the return value. If you do not want to use assertEquals, which would require the return value match the expected value exactly, you could use assertTrue and check that the return value contains the expected value. For example, for Quatrefoil example3:

```
Assert.assertTrue(example3.toString().contains("\"Large  Example\""));
```

Also, 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. You can also step-in to the method being called by the test method and then single-step through it, looking for the error.

- **QuatrefoilList.java** (new for Part 3) – Consider implementing this file in parallel with its test file, QuatrefoilListTest.java, which is described after this class.

  Requirements: Create a QuatrefoilList class that stores the name of the list and an array of Quatrefoil objects. It also includes methods that return the name of the list, number of Quatrefoil objects in the QuatrefoilList, total area and average area for all Quatrefoil objects in the QuatrefoilList. The toString method returns summary information about the list (see below).

  Design: The QuatrefoilList 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 Quatrefoil objects, and (3) an `int` representing the number of Quatrefoil objects in the array, which may be less than the length of the array of Quatrefoil objects. These instance variables should be private so that they are not directly accessible from outside of the QuatrefoilList class. These should be the only fields (or instance variables) in this class, and they should be initialized in the constructor described below.

  (2) Constructor: Your QuatrefoilList class must contain a constructor that accepts three parameters: (1) a parameter of type String representing the name of the list, (2) a parameter of type `Quatrefoil[]`, representing the list of Quatrefoil objects, and (3) a parameter of type `int` representing the number of Quatrefoil objects in the array. These parameters should be used to assign the fields described above (i.e., the instance variables).

  (3) Methods: The methods for QuatrefoilList are described below.
    - `getName`: Returns a String representing the name of the list.
    - `numberOfQuatrefoils`: Returns an int (the value of the third field in the QuatrefoilList object) representing the number of Quatrefoil objects in the QuatrefoilList.
    - `totalDiameters`: Returns a double representing the total diameters in inches for all Quatrefoil objects in the list. If there are zero Quatrefoil objects in the list, zero should be returned.
    - `totalPerimeters`: Returns a double representing the total perimeters in inches for all Quatrefoil objects in the list. If there are zero Quatrefoil objects in the list, zero should be returned.

- `totalAreas`: Returns a double representing the total areas in square inches for all Quatrefoil objects in the list. If there are zero Quatrefoil objects in the list, zero should be returned.
- `averageDiameter`: Returns a double representing the average diameter for all Quatrefoil objects in the list. If there are zero Quatrefoil objects in the list, zero should be returned.
- `averagePerimeter`: Returns a double representing the average perimeter in inches for all Quatrefoil objects in the list. If there are zero Quatrefoil objects in the list, zero should be returned.
- `averageArea`: Returns a double representing the average area in square inches for all Quatrefoil objects in the list. If there are zero Quatrefoil objects in the list, zero should be returned.
- `toString`: Returns a String (does not begin with \n) containing the name of the list (which can change depending on the name of the list passed as a parameter to the constructor) followed by various summary items: number of Quatrefoils, total diameters, total perimeters, total areas, average diameter, average perimeter, and average area. Use "#,##0.0##" as the pattern to format the double values. Below is an example of the formatted String returned by the `toString` method, where the name of the list (name field) is `Quatrefoil Test List` and the array of Quatrefoil objects contains the three examples described above (bottom of page 2 and top of page 3).
```
----- Summary for Quatrefoil Test List -----
Number of Quatrefoils: 3
Total Diameters: 153.34 inches
Total Perimeters: 481.732 inches
Total Areas: 8,933.417 square inches
Average Diameter: 51.113 inches
Average Perimeter: 160.577 inches
Average Area: 2,977.806 square inches
```
- `getList`: Returns the array of Quatrefoil objects (the second field above).
- `addQuatrefoil`: Returns nothing but takes two parameters (label and diameter), creates a new Quatrefoil object, and adds it to the QuatrefoilList object in the next available location in the Quatrefoil array. Be sure to increment the int field containing the number of Quatrefoil objects in the QuatrefoilList object.
- `findQuatrefoil`: Takes a label of a Quatrefoil as the String parameter and returns the corresponding Quatrefoil object if found in the QuatrefoilList object; otherwise returns null. Case should be ignored when attempting to match the label.
- `deleteQuatrefoil`: Takes a String as a parameter that represents the label of the Quatrefoil and returns the Quatrefoil if it is found in the QuatrefoilList object and deleted; otherwise returns null. Case should be ignored when attempting to match the label. 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 Quatrefoil element in the array should be set to null. Finally, the number of elements field must be decremented.
- `editQuatrefoil`: Takes two parameters (label and diameter), uses the label to find the corresponding the Quatrefoil object in the list. If found, sets the diameter field to the value passed in as a parameter, and returns true. If not found, returns false. (*Note that the label should not be changed by this method*.)

- o `findQuatrefoilWithLargestDiameter`: Returns the Quatrefoil with the largest diameter; if the list contains no Quatrefoil objects, returns null.

Code and Test: Some of the methods above require that you use a loop to go through the objects in the array. You should 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.

- **QuatrefoilListTest.java** (<u>new for Part 3</u>) – Consider implementing this test file in parallel with the its corresponding source file, QuatrefoilList.java, which is described above this class.

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

  Design: Typically, in each test method, you will need to create an instance of QuatrefoilList, 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 QuatrefoilList. <u>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</u>. 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 the methods in your QuatrefoilList class.

  Code and Test: A good strategy would be to begin by writing test methods for those methods in QuatrefoilList 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 QuatrefoilList 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 QuatrefoilList. Be sure to call the QuatrefoilList toString method in one of your test cases so that the grading system will consider the toString method to be "covered" in its coverage analysis. Remember that when a test method fails, 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. You can also step-in to the method being called by the test method and then single-step through it, looking for the error.

  <u>Finally, 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</u>

the equals method implemented in Part 2.

The Grading System

When you submit your files (Quatrefoil.java, QuatrefoilTest.java, QuatrefoilList.java, and QuatrefoilListTest.java), the grading system 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. In this project, your test files should provide method, statement, and condition coverage. Each condition in your source file must be exercised both true and false. See below for a description of how to test a boolean expression with multiple conditions. To see your code coverage in Web-CAT, on "Your Assignment Submission Results" page, in "File Details", click on your source file and look for any highlighted lines of code. Hover your mouse on the highlighted source code, to see the hint about what was not covered. Example of hints you might see include: (1) This line was never executed by your tests. (2) Not all possibilities of this decision were tested. Remember that when you have N simple conditions combined, you must test all N+1 possibilities.

Note For Testing the `equals` Method (If you successfully did this in Part 2, you may ignore this Note.)

Perhaps the most complicated method to test is the `equals` method in Quatrefoil. Assuming the parameter is an instance of Quatrefoil, this method has two conditions in the boolean expression that are &&'d in the 'else' block. Since Java (and most other languages) uses short-cut logic, if the first condition in an && is false, the &&'d expression is false (i.e., the next condition is not checked). This means that to test the second condition, the first conditions must be true. To have condition coverage for the equals method, you need the three test cases where the two conditions evaluate to the following, where T is true, F is false, and X is don't care (could be true or false):

FX - returns false
TF - returns false
TT - returns true