Unit 2 Lab: Arrayd calculator

Objectives:

To practice breaking problems into subtasks, writing and calling methods, and use the `DecimalFormat` class.

Background

In the last lab, you wrote a program for a space merchant establishing a trading network with the population of the planet Arrayd. Since humans have made contact with the Arrayians, they have taught the Arrayians how to translate from Array to English and the decimal (base-10) system to represent numbers (both integer and non-integer values). The Arrayians have started learning about human mathematical operations. The Arrayians were so impressed with your first program, they would like you to build a program that helps them use human math to do Arrayian math. The program will perform Arrayian math using human mathematical operators.

The laws of nature on planet Arrayd are not the same as on Earth, so Arrayian mathematical operations are different. For example, there is gravity on Arrayd, but the force of gravity increases as altitude increases.

Arrayians do not use longitude and latitude to locate places on Arrayd. Instead, all locations are represented by a coordinate consisting of 3 values. The first number is called the k-value, a value relating the distance from the location to the leader's castle. The second number is called the m-value, a value relating the distance from the location to the Register0 Monument. The third number is called the b-value, a value relating the distance from the location to the CPU Building in the capital city. All locations can be uniquely described using the coordinate (k,m,b).

Assignment

Complete the program in the shell provided to you. **Do not change any of the code already in the shell. Submissions that contain changes to the test code will be given a 50%.** Add your methods in the `ArrayianCalculator` class below the main method. You may only use material from Units 1 and 2 and the `DecimalFormat` class to write your program. Submissions that contain material from other units or other Java libraries will be given a 50%.

You must write the 9 methods with the bolded given names exactly as described below in order for your program to pass the included test cases. Do not repeat code that has been written in a method elsewhere in the class. Call the necessary methods instead. Submissions that contain repeated code instead of method calls will be given a 50%. ***You must also write preconditions and postconditions for each method in a comment above the method header.***

Methods that return the value of Arrayian operations, that is the methods use human (Java) operators to perform Arrayian operations:

- **add**: this method returns the result of Arrayian addition on 2 floating point values passed into the parameters. Arrayian addition is commutative (a $\text{add}_{\text{arrayian}}$b = b $\text{add}_{\text{arrayian}}$ a). Arrayian addition is calculated with the human operations: add together the sum of the 2 values and the product of the 2 values.
- **subtract**: this method returns the result of Arrayian subtraction on 2 floating point values passed into the method. Arrayian subtraction is commutative (a $\text{subtract}_{\text{arrayian}}$ b = b $\text{subtract}_{\text{arrayian}}$ a). Arrayian subtraction is calculated with the human operations: subtract the larger value and the smaller value (large minus small). Multiply this difference by the absolute value of the smaller parameter.
- **multiply**: this method returns the result of Arrayian multiplication on 2 floating point values passed into the parameters. Arrayian multiplication is commutative (a $\text{multiply}_{\text{arrayian}}$ b = b $\text{multiply}_{\text{arrayian}}$ a). Arrayian multiplication is calculated with the human operations: multiply the values and triple this product.
- **divide**: this method returns the result of Arrayian division on 2 floating point values passed into the parameters so that the first parameter is divided by the second parameter. Arrayian division is not commutative (a $\text{divide}_{\text{arrayian}}$ b ≠ b $\text{divide}_{\text{arrayian}}$ a). The result of Arrayian division is always positive. Arrayians also think division by 0 is stupid, so it cannot be performed when the second parameter's value is 0. Arrayian division is calculated with the human operations: divide the first parameter by the second parameter. Square root the absolute value of this quotient. A user should never call this method with a second argument of 0.
- **squareRoot**: this method returns the Arrayian square root of 1 floating point value passed into the method. The Arrayian square root is the human fourth root ($\sqrt[4]{n}$) operation. As a result, the square root cannot be performed on values less than 0. A user should never call this method with a negative argument.

Methods written using **ONLY** Arrayian operations:

- **square:** this method returns the Arrayian square (second power) of 1 floating point value passed into the method. The Arrayian square has the same *definition* as the human square operation! The square of a value is the value multiplied by itself!
- **calcDistance**: this method will return the distance between 2 Arrayian locations. The coordinates of the locations are passed into the 6 floating point parameters of the method. The first three parameters represent the coordinate of the first location, in the proper order (k_0,m_0,b_0). The last three parameters represent the coordinate of the second location, in the proper order (k_1,m_1,b_1). Subtract the corresponding coordinate values (subtract k-values, subtract m-values, subtract b-values). Square each of these differences. Add the 3 differences together. Then square root the sum.
- **calcTrianglePerimeter**: this method will return the perimeter of an Arrayian triangle with the coordinates of the triangle's three vertices passed into the nine floating point parameters of this method (k_0,m_0,b_0,k_1,m_1,b_1, k_2,m_2,b_2). The perimeter is the sum of the 3 distances between the adjacent vertices.

- **printMessage**: this method will not return a value. It will print a message to the console about the Arrayian sum, difference, product, and quotient of the 2 floating point values passed in to the parameters. You will need to use the `DecimalFormat` class as described below. For example, the method call `printMessage(2.5,8)` would print:

```
Using the two values: 2.5 and 8.0
The sum is: 30.50
The difference is: 13.75
The product is: 400.00
The quotient is: 0.56
```

DecimalFormat class

Programmers have many options to represent floating point values. An option is to use the Java class `DecimalFormat`. You need to create an instance from a Java class that has already been written, `DecimalFormat`. This is like the `Scanner` objects you create to accept user input.
First, you will need to include an import statement:

```
import java.text.DecimalFormat
```

To create an instance of the class, use:

```
DecimalFormat d  = new DecimalFormat (“0.00”);
```

To format a value using this instance, use the method call:

```
d.format(number or numeric variable here)
```

For example, to format the number 1234.56789 to 1234.57 (it is rounded) after creating an instance of the `DecimalFormat` class named `money`, you would call `money.format(1234.56789)`. If that value were stored in a variable named `cost`, you would call: `money.format(cost)`. The value returned is a `StringBuffer` object (characters), not a numeric amount. You can use this expression almost anywhere you would use a `String`, such as in a `print` statement or with the `+` operator.

For example, if you wanted to format the variable `nearestThous` holding an amount to three decimal places and then print the value, you would write the following code:

```
DecimalFormat d = new DecimalFormat("0.000");
System.out.println("This is the value rounded to the nearest thousandth: " + d.format(nearThous));
```

You must write the code to call the `format` method in the `printMessage` method.

Tolerance testing

Floating point values are inherently imprecise because storing base-10 real (non-integer) numbers using base-2 is impossible in many cases. For example, you cannot precisely represent tenths (0.1) using only multiples of powers of 2 ($2^{-1} = 0.5$, $2^{-2} = 0.25$, $2^{-3} = 0.125$, etc.). Tolerance testing is one of the ways to determine if floating point calculations are performing properly even though the answer produced by 2 different sets of code may not be exactly the same. The amount that answers are allowed to vary by is the tolerance.

For example, if you multiplied two floating point values and expected the answer to be 9.12340 but would allow for answers in the range 9.12335 - 9.12345 due to rounding error, you could set the expected result to be 9.12340 and the tolerance to be 0.000005. You could then write a test that accepts these answers with the code:

```
double calculatedAmount = //whatever the calculation is
double tolerance = 0.000005;
double expectedVal = 9.1234;
double difference = Math.abs(calculatedAmount-expectedVal);   //difference between expected amount
                                                               //and actual result of calculation
if (difference<tolerance)
   System.out.println("test passed!!");
```

Tolerance testing is used in the code for the test cases. Check it out! You may have to use it in your code one day.

Expected Output
Use the test cases to determine the expected output for each method!

Grading:
This lab is worth 10 points based on the following criteria:

- Your program passes the 8 test cases and follows all instructions in this lab (8 pts)
- Your program follows good programming style guidelines (1 pt)
- Correct precondition and postconditions above every method header (1 pt)