The City College of New York, CUNY
Department of Compute Science
North Academic Center, Room 8/206
160 Convent Avenue, New York, NY 10031

Assignment 4 – Fall 2020

Due Date: *by Sunday November 8, 2020 11:59PM*
How to submit: *compress and upload JAVA files to Blackboard*

Network Log Utility.

In this assignment we implement a Java class hierarchy which represents various two-dimensional and three-dimensional shapes. The hierarchy extends classes/interfaces from Java's API and defines new abstract and concrete classes. As a start, I am providing the class' stubs without any implementation. You may choose to write your implementation without relying on the provided stubs; however, please adhere to the naming conventions shown in the UML diagram. A test class is provided for this assignment. You will be graded on the efficiency of your code; reuse parent implementation whenever possible. Please submit a ZIP file containing your 6 classes.

Note:

- ✓ ***This is an individual assignment; please do your own work, sharing and/or copying code and/or solution ideas with/from others will result in a grade of 0 and disciplinary actions for all involved parties. If you run into problems and have done your best to solve them, please contact me before/after class or by e-mail.***
- ✓ ***A 20% grade deduction for every day the assignment is late.***

Preamble

From Java's API we will utilize a number of classes including:

I) $TreeSet$, a class for maintaining a collection of objects in a sorted fashion. Using $TreeSet$ is similar to using an $ArrayList$. The $TreeSet$ works best with objects that implement the $Comparable$ interface.

II) $Comparable$, a generic type interface (i.e. template); classes that implement this interface are forced to implement the method $compareTo$. The $compareTo(\ )$ defines how two objects of the same type are compared. The basic implementation of the $compareTo(\ )$ is to, first, decide on how to compare two objects and, second, return either:

- ✓ A 0 if two objects are deemed equivalent.
- ✓ A negative number if the left object is smaller than the right object
- ✓ A positive number if the left object is greater than the right object

Here is an example which compares two class objects based on the $first$ and $second$ fields respectively. Two objects of type $MyClass$ are deemed equal if their $first$ and $second$ values are the same. If the $first$ fields are not equal, the result of $compareTo(\ )$ are based on comparing the $first$ fields only. However, if the $first$ fields are equal, the result of $compareTo(\ )$ is based on the comparison between the $second$ fields.

```
public class MyClass implements Comparable<MyClass> { // note the class name between angle brackets
    int first;
    String second;
    @Override
    public int compareTo(MyClass other) {
        int result = Integer.compare(first, other.first);
        if (result == 0) // if the first fields match, results are based on comparing the second fields
            return second.compareTo(other.second); // result is based on comparing the second fields

        return result;      // here when the first fields do not match
    }
}
```

The City College of New York, CUNY
Department of Compute Science
North Academic Center, Room 8/206
160 Convent Avenue, New York, NY 10031

Class' Description:

The assignment consists of 6 Java classes, 3 abstract and 3 non-abstract classes and 1 test class. Class stubs are provided for an easier start.

I. *Shape*:

- ✓ An abstract class which implements Java's $Comparable$ Interface
- ✓ Contains two abstract methods, $area$ and $perimeter$.
- ✓ $toString$: returns a space-delimited string of its fields:
  $$\ll value\ of\ id \gg \ll value\ of\ name \gg \ll value\ of\ description \gg \ll value\ of\ color \gg$$
- ✓ $getColorName$: returns the name of the $Color$ as a $String$. This method is simply the reverse implementation of $getColor$ in the test class
- ✓ $compareTo$: numerically compares two objects of type $Shape$. Returns the value 0 if both objects have the same $name$ and $color$. Use $getColorName(\ )$ when comparing colors. Note that the id and $description$ fields are excluded here. Otherwise, return the results of the first mismatch comparison between $name$ then $color$. In other words, if the two $name$ fields are the same return the results of comparing the two $color$ fields.

II. *Shape2D* and *Shape3D*:

- ✓ Abstract classes which inherits from class $Shape$.
- ✓ The non-default constructor initializes the class' private fields
- ✓ $toString$: returns the same value described in the parent class but includes $height$, $width$, and $length$ fields. The method must re-use the parent class' $toString$ implementation.
- ✓ $compareTo$ numerically compares two objects of type $Shape2D$ (or $Shape3D$). Returns the value 0 if both objects have the same id, $name$, $description$, $width$, $height$, and $length$ (for $Shape3D$ only). The $compareTo(\ )$ in $Shape2D$ must re-use the $compareTo(\ )$ from the $Shape$ class. The $compareTo(\ )$ in $Shape3D$ must reuse the $compareTo(\ )$ from the $Shauupe2D$ class.

III. *Quadrilateral*

- ✓ Represents 90° angle quadrilateral 2D shapes
- ✓ Inherits from class $Shape$
- ✓ $area = \text{width} \times \text{height}$
- ✓ $perimeter = 2 \times (\text{width} + \text{height})$

IV. *Quadrilateral3D*

- ✓ Represents 90° angle quadrilateral 3D shapes
- ✓ $area = 2 \times (\text{width} \times \text{height} + \text{width} \times \text{length} + \text{height} \times \text{length})$
- ✓ $perimeter = 4 \times (\text{width} + \text{height} + \text{length})$

V. *ShapeList*

- ✓ Extends $java.util.TreeSet$
- ✓ add: checks if a similar $Shape$ instance is already stored. If a similar object is found, the method returns false. If it is not, the object is added and the method returns $true$. YOU MUST USE the $contains(\ )$ method from $TreeSet$ which requires that the method $compareTo(\ )$ is be overloaded properly. Do not write your own search code.
- ✓ $get2DShapes$: returns a new set containing instances of supertype $Shape2D$ ONLY.
  Hint: the $instaceof$ operator is useful here.
- ✓ $get3DShapes$: returns a new set containing instances of supertype $Shape3D$ ONLY.
- ✓ $printFormatted$: prints a sorted and formatted table of all $Shape$ objects (Figure 2). The list is automatically sorted by $Name$, then $Color$, then $Dimension$ based on the $compareTo(\ )$ implementation described earlier.

VI. *A4Test*:

- ✓ This is the provided test class. Your code should work with this class AS IS. Please DO NOT modify or submit this class and adhere to the names provided in the class' UML diagram (Figure 1). You may, however, comment lines of code until you are ready to test the methods they invoke.
- ✓ Your code's output should match the output shown in Figure 2

Grading:

Item	Points
Class Shape	10
Class Shape2D	10
Class Shape3D	10
Class Quadrilateral	10
Class Quadrilateral3D	10
Class ShapeList	
add	10
get2DShapes and get3DShapes	10
printFormatted	10
Correct output	10
Efficiency of code	10
	100

Figures:

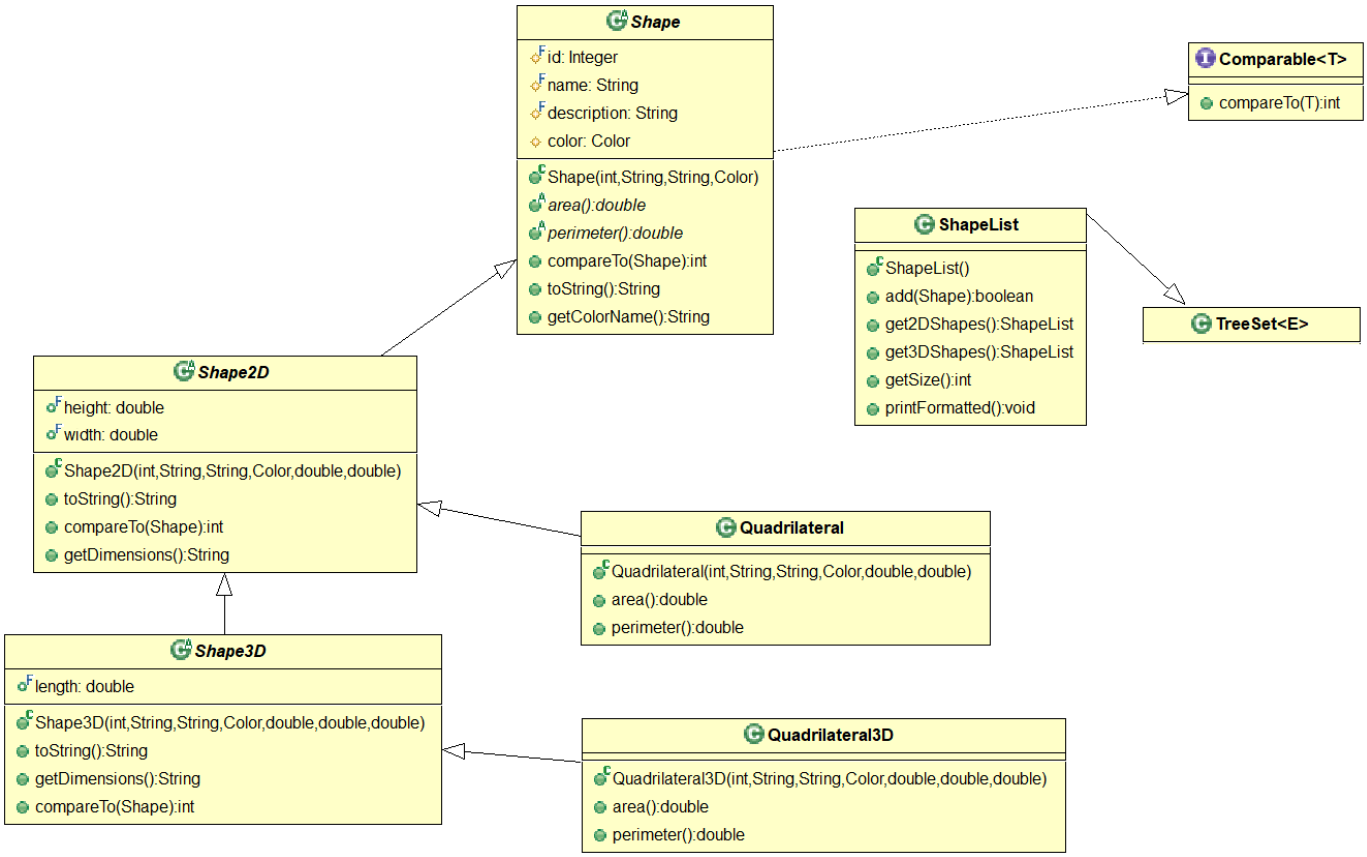

Figure 1: Class UML Diagram

```
java.lang.UnsupportedOperationException: Unrecognized shape, skipping: 98241,BLACK,18.785:15.059,Sphere,A black sphere
java.lang.UnsupportedOperationException: Unrecognized shape, skipping: 57057,BLACK,7.85:,Circle,A black sphere
The list contains 29 Shape objects
There are 18 2-Dimensional shapes
There are 11 3-Dimensional shapes
```

ID	Name	Color	Dimensions	Description
35779	Cube	Blue	39.20 X 46.91 X 27.15	A blue cube
90238	Cube	Blue	77.60 X 54.66 X 8.34	A blue cube
65701	Cube	Red	90.33 X 56.78 X 44.61	A red cube
40433	Cube	Yellow	94.89 X 43.88 X 21.47	A yellow cube
51060	Cuboid	Black	43.61 X 94.74 X 65.44	A black cuboid
90955	Cuboid	Blue	55.69 X 40.01 X 90.70	A blue cuboid
83912	Cuboid	Blue	8.09 X 45.01 X 96.79	A blue cuboid
64851	Cuboid	Green	98.18 X 5.51 X 64.22	A green cuboid
88174	Cuboid	Green	94.47 X 6.70 X 83.56	A green cuboid
37951	Cuboid	Red	70.19 X 32.90 X 94.36	A red cuboid
36830	Cuboid	Yellow	5.38 X 59.99 X 69.06	A yellow cuboid
48900	Rectangle	Red	57.56 X 4.03	A red rectangle
60665	Rectangle	Red	77.08 X 60.12	A red rectangle
90965	Rectangle	Red	94.77 X 61.20	A red rectangle
72916	Rectangle	White	24.37 X 86.85	A white rectangle
35886	Rectangle	Yellow	65.95 X 51.71	A yellow rectangle
60895	Square	Black	87.86 X 39.68	A black square
67132	Square	Black	33.89 X 83.52	A black square
44356	Square	Blue	57.32 X 54.03	A blue square
85368	Square	Cyan	70.95 X 41.82	A cyan square
99999	Square	Cyan	61.01 X 44.17	A cyan square
26449	Square	Cyan	84.56 X 77.16	A cyan square
71002	Square	Green	23.65 X 12.55	A green square
78853	Square	Red	7.28 X 60.85	A red square
37376	Square	Red	63.30 X 91.52	A red square
25280	Square	White	5.54 X 0.04	A white square
66544	Square	White	43.07 X 30.90	A white square
27982	Square	White	29.25 X 88.76	A white square
76667	Square	Yellow	38.50 X 62.44	A yellow square

Figure 2: Test Class' Output

The City College of New York, CUNY
Department of Compute Science
North Academic Center, Room 8/206
160 Convent Avenue, New York, NY 10031

UML Diagram Legend

Symbol	Description
Underlined	Indicates a static member
	A private member (i.e. variable or method)
	A private final member (i.e. variable)
	A public field (i.e. variable or method)
	A public abstract member (i.e. variable or method)
	A public constructor
	A static public member
	An interface
	A public class
	A public abstract class
	A hollowed arrow indicates inheritance
	An open-ended arrow indicates composition
	A dotted line and hollowed arrow indicate class implementation