**Software Developer Course Assessment**

**Quantitative Assessment Practice**

**Course Name: Advanced Programming (Java)**

**Current Week: 7th November 2024**

**Submission date:21st November 2024**

**Introduction:**

The purpose of this assessment is to help us understand how the class is doing in terms of the review material that we have covered during the previous couple of weeks. The **only** purpose of this assessment is for us to improve our approach to review and ensure that what we’re currently doing is an effective strategy. Completion of this assessment is **mandatory - if you don’t submit a solution, it will be marked as incomplete. You must complete a minimum of 80% of your assigned QAPs per course – otherwise you will be marked as incomplete for that course no matter how good your other grades are.** If you do submit a solution, it will be marked as complete, as you will receive full marks no matter what your actual performance was – again this is a participation grade.  
  
Again, the goal here is to help you all in the best way that we can, so please do be honest when answering the questions related to how long it took, which resources you used, etc. And please ensure that you do your **own** work – don't just copy off a friend to get it done, earnestly do your best with it. If you can’t get it completely working, give us what you have. While it will be graded, the grade will not count against you, it’s just a way for us to see where everybody is, and to know which concepts, if any, we, as a class, may be struggling with.

**Deadline:** You will have until the end of the day on , 21st November (11:59pm) to submit your assessment solutions. Please ensure you answer all the questions outlined in the instructions portion of this document as well in your submission.

**Marking:** In this program core evaluation is marked with one of three possible marks: *Incomplete, Pass, Pass Outstanding.* For QAPs, though, where incomplete marks are more important for our own information as well as for the information of the student, we wanted to increase the resolution of our grading system. Therefore, QAPs are marked on a scale of 1-5. The details of this marking system are summarized in the table below.

|  |  |
| --- | --- |
| **Grade** | **Meaning** |
| 1 | *Incomplete.* Student shows severe lack of understanding of the material – solution is heavily incomplete, non-functional, or completely off base of what the assignment was asking for. |
| 2 | *Partially Complete.* Students show some understanding of the material. Solution may be non-functional or partially functional, but the approach is correct, albeit with some major bugs or missing features. |
| 3 | *Mostly Complete.* Student demonstrates understanding of the major ideas of the assignment. Solution is mostly working, albeit with a few small bugs or significant edge cases which were not considered. Shows a good understanding of the correct approach, and is either nearly a feature-complete solution, or is a feature-complete solution with some bugs. |
| 4 | *Complete (Equivalent to: Pass.)* Student shows complete understanding of assigned work and implemented all necessary features. Any bugs that are present are insignificant (for example aesthetic bugs when testing the functionality of code) and do not impact the core functionality in a significant way. All necessary objectives for the assignment are completed, and the student has delivered something roughly equivalent to the canonical solution in terms of features and approach. |
| 5 | *Complete with Distinction (Equivalent to: Pass Outstanding)* The student demonstrates a clear mastery of the subject matter tested by the QAP. The solution goes above and beyond in some way, makes improvements on the canonical solution, or otherwise demonstrates the student’s mastery of the subject matter in some way. A solution in this category would consider all reasonable edge cases and implement more than the necessary functionality required by the assignment. |

**Instructions:**

**You are allowed to complete the assessment problems below in whatever way you can but please answer the following questions/points as part of your submission:**

1. How many hours did it take you to complete this assessment? (Please keep try to keep track of how many hours you have spent working on each individual part of this assessment as best you can - an estimation is fine; we just want a rough idea.)

[It took me around 23 hours on all of them. Most of it went to solving 3rd and 4th problems]

1. What online resources you have used? (My lectures, YouTube, Stack overflow etc.)

[I did use lectures, W3School, and asked some questions on the random websites. Used ChatGPT sometimes to verify code or when could not find the problem of it not working]

1. Did you need to ask any of your friends in solving the problems. (If yes, please mention name of the friend. They must be amongst your class fellows.)

[No I did not]

1. Did you need to ask questions to any of your instructors? If so, how many questions did you ask (or how many help sessions did you require)?

[Did not ask teachers any questions. But had a moment when really wanted to ask about 3rd or 4th problem, but solved them before the next session.]

1. Rate (subjectively) the difficulty of each question from your own perspective, and whether you feel confident that you can solve a similar but different problem requiring some of the same techniques in the future now that you’ve completed this one.

[I am really confident in first problem, will easily solve it. Second get me to think a little bit, but still easily solved. Third and fourth were the hardest. Got me thinking a lot. If not for the files that need to be submitted, I would think for much longer. I do not think that I will be able to solve similar yet, if I would not have more steps defined in the creation.]

**Problem#1:**

A HighSchool application has two classes: the Person superclass and the Student subclass. Using inheritance, in this lab you will create two new classes, Teacher and CollegeStudent. A Teacher will be like Person but will have additional properties such as *salary* (the amount the teacher earns) and *subject* (e.g. “Computer Science”, “Chemistry”,  “English”, “Other”). The CollegeStudent class will extend the Student class by adding a *year*(current level in college) nd *major* (e.g. “Electrical Engineering”, “Communications”, “Undeclared”).

The inheritance hierarchy would appear as follows:

Diagram

Description automatically generatedListed below is the Person base class to be used as a starting point for the Teacher class:

**class** Person {

**protected** String myName ;   // name of the person  
**protected** **int** myAge;        // person’s age  
**protected** String myGender;  // “M” for male, “F” for female

**public** Person(String name, **int** age, String gender)  {

myName = name; myAge = age ; myGender = gender;   }

**public**String toString()  {  
**return** myName + “, age: ” + myAge +“, gender: ” +myGender;  
}  
}

The Student class is derived from the Person class and used as a starting point for the CollegeStudent class:

**class** Student **extends** Person {  
**protected** String myIdNum;    // Student Id Number  
**protected** double myGPA;      // grade point average

**public** Student(String name, **int** age, String gender,String idNum, **double** gpa)  {  
// use the super class’ constructor  
**super**(name, age, gender);

// initialize what’s new to Student

myIdNum = idNum;

myGPA = gpa;

}     }

**Tasks**:

1. Add methods to “set” and “get” the instance variables in the Person class. These would consist of: getName, getAge, getGender, setName,  setAge, and setGender.

2. Add methods to “set” and “get” the instance variables in the Student class. These would consist of: getIdNum, getGPA, setIdNum, and setGPA.

3. Write a Teacher class that extends the parent class Person.

a.  Add instance variables to the class for *subject* (e.g. “Computer Science”, “Chemistry”,, “English”, “Other”) and*salary* (the teachers annual salary). *Subject* should be of type String and *salary* of type double. Choose appropriate names for the instance variables.

b.  Write a constructor for the Teacher class. The constructor will use five parameters to initialize myName, myAge, myGender, *subject*, and *salary*.  Use the super reference to use the constructor in the Person superclass to initialize the inherited values.

c.  Write “setter” and “getter” methods for all of the class variables. For the Teacher class they would be: getSubject, getSalary, setSubject, and setSalary.

d.  Write the toString() method for the Teacher class. Use a super reference to do the things already done by the superclass.

4. Write a CollegeStudent subclass that extends the Student class.

a.   Add instance variables to the class for *major* (e.g. “Electrical Engineering”, “Communications”, “Undeclared”) and *year* (e.g. FROSH = 1,  SOPH = 2, …). *Major* should be of type String and *year* of type int. Choose appropriate names for the instance variables.

b. Write a constructor for the CollegeStudent class. The constructor will use seven parameters to initialize myName, myAge, myGender,  myIdNum, myGPA, *year*, and *major*. Use the super reference to use the constructor in the Student superclass to initialize the inherited values.

c. Write “setter” and “getter” methods for all of the class variables. For the CollegeStudent class they would be: getYear, getMajor, setYear,  and setMajor.

d. Write the toString() method for the CollegeStudent class. Use a super reference to do the things already done by the superclass.

5. Write a testing class with a main() that constructs all of the classes (Person, Student, Teacher, and CollegeStudent) and calls their toString()  method.  Sample usage would be:

Person bob = **new** Person(“Coach Bob”, 27, “M”);  
System.out.println(bob);

Student lynne = **new** Student(“Lynne Brooke”, 16, “F”, “HS95129″, 3.5);  
System.out.println(lynne);

Teacher mrJava = **new** Teacher(“Duke Java”, 34, “M”, “Computer Science”, 50000);|  
System.out.println(mrJava);

CollegeStudent ima = **new** CollegeStudent(“Ima Frosh”, 18, “F”, “UCB123″, 4.0, 1, “English”);

System.out.println(ima);

**Deliverables:**

**Complete and working-class files with proper comments.**

1. **Person.java**

package Problem\_one;

public class Person {

protected String myName; // Name of the person

protected int myAge; // Person's age

protected String myGender; // "M" for male, "F" for female

// Constructor

public Person(String name, int age, String gender) {

myName = name;

myAge = age;

myGender = gender;

}

// Getters

public String getName() {

return myName;

}

public int getAge() {

return myAge;

}

public String getGender() {

return myGender;

}

// Setters

public void setName(String name) {

myName = name;

}

public void setAge(int age) {

myAge = age;

}

public void setGender(String gender) {

myGender = gender;

}

// toString method

public String toString() {

return myName + ", age: " + myAge + ", gender: " + myGender;

}

}

1. **Student.java**

package Problem\_one;

public class Student extends Person {

protected String myIdNum; // Student ID Number

protected double myGPA; // Grade Point Average

// Constructor

public Student(String name, int age, String gender, String idNum, double gpa) {

super(name, age, gender); // Initialize Person attributes

myIdNum = idNum;

myGPA = gpa;

}

// Getters

public String getIdNum() {

return myIdNum;

}

public double getGPA() {

return myGPA;

}

// Setters

public void setIdNum(String idNum) {

myIdNum = idNum;

}

public void setGPA(double gpa) {

myGPA = gpa;

}

// toString method

public String toString() {

return super.toString() + ", ID: " + myIdNum + ", GPA: " + myGPA;

}

}

1. **Teacher.java**

package Problem\_one;

public class Teacher extends Person {

private String subject; // Subject taught by the teacher

private double salary; // Teacher's annual salary

// Constructor

public Teacher(String name, int age, String gender, String subject, double salary) {

super(name, age, gender); // Initialize Person attributes

this.subject = subject;

this.salary = salary;

}

// Getters

public String getSubject() {

return subject;

}

public double getSalary() {

return salary;

}

// Setters

public void setSubject(String subject) {

this.subject = subject;

}

public void setSalary(double salary) {

this.salary = salary;

}

// toString method

public String toString() {

return super.toString() + ", subject: " + subject + ", salary: $" + salary;

}

}

1. **CollegeStudent.java**

package Problem\_one;

public class CollegeStudent extends Student {

private String major; // Major of the College Student

private int year; // Current year in college (1 = Freshman, etc.)

// Constructor

public CollegeStudent(String name, int age, String gender, String idNum, double gpa, int year, String major) {

super(name, age, gender, idNum, gpa); // Initialize Student attributes

this.year = year;

this.major = major;

}

// Getters

public int getYear() {

return year;

}

public String getMajor() {

return major;

}

// Setters

public void setYear(int year) {

this.year = year;

}

public void setMajor(String major) {

this.major = major;

}

// toString method

public String toString() {

return super.toString() + ", year: " + year + ", major: " + major;

}

}

1. **Demo.java**

package Problem\_one;

public class Demo {

public static void main(String[] args) {

// Testing Person

Person bob = new Person("Coach Bob", 27, "M");

System.out.println(bob);

// Testing Student

Student lynne = new Student("Lynne Brooke", 16, "F", "HS95129", 3.5);

System.out.println(lynne);

// Testing Teacher

Teacher mrJava = new Teacher("Duke Java", 34, "M", "Computer Science", 50000);

System.out.println(mrJava);

// Testing CollegeStudent

CollegeStudent ima = new CollegeStudent("Ima Frosh", 18, "F", "UCB123", 4.0, 1, "English");

System.out.println(ima);

}

}

1. **Screenshot of the running code’s output**

**Coach Bob, age: 27, gender: M**

**Lynne Brooke, age: 16, gender: F, ID: HS95129, GPA: 3.5**

**Duke Java, age: 34, gender: M, subject: Computer Science, salary: $50000.0**

**Ima Frosh, age: 18, gender: F, ID: UCB123, GPA: 4.0, year: 1, major: English**

**Problem#2:**

Write the classes as shown in the following class diagram. Also write a demo class with the main method, to show the working of the application. Mark all the overridden methods with annotation @Override.

Table

Description automatically generated

**Hints**

1. You cannot assign floating-point literal say 1.1 (which is a double) to a float variable, you need to add a suffix f, e.g. 0.0f, 1.1f.
2. The instance variables x and y are private in Point and cannot be accessed directly in the subclass MovablePoint. You need to access via the public getters and setters. For example, you cannot write x += xSpeed, you need to write setX(getX() + xSpeed).

**Deliverables:**

**Complete and working-class files with proper comments.**

1. **Point.java**

package Problem\_two;

/\*\*

\* Represents a Point in 2D space.

\*/

public class Point {

private float x = 0.0f; // X-coordinate

private float y = 0.0f; // Y-coordinate

// Default constructor

public Point() {

}

// Parameterized constructor

public Point(float x, float y) {

this.x = x;

this.y = y;

}

// Getter for x

public float getX() {

return x;

}

// Setter for x

public void setX(float x) {

this.x = x;

}

// Getter for y

public float getY() {

return y;

}

// Setter for y

public void setY(float y) {

this.y = y;

}

// Sets both x and y coordinates

public void setXY(float x, float y) {

this.x = x;

this.y = y;

}

// Returns x and y coordinates as an array

public float[] getXY() {

return new float[]{x, y};

}

// toString method

@Override

public String toString() {

return "(" + x + ", " + y + ")";

}

}

1. **MovablePoint.java**

package Problem\_two;

/\*\*

\* Represents a MovablePoint in 2D space, which can move based on speed.

\*/

public class MovablePoint extends Point {

private float xSpeed = 0.0f; // Speed in the X direction

private float ySpeed = 0.0f; // Speed in the Y direction

// Default constructor

public MovablePoint() {

super();

}

// Constructor with speeds only

public MovablePoint(float xSpeed, float ySpeed) {

super();

this.xSpeed = xSpeed;

this.ySpeed = ySpeed;

}

// Constructor with coordinates and speeds

public MovablePoint(float x, float y, float xSpeed, float ySpeed) {

super(x, y);

this.xSpeed = xSpeed;

this.ySpeed = ySpeed;

}

// Getter for xSpeed

public float getXSpeed() {

return xSpeed;

}

// Setter for xSpeed

public void setXSpeed(float xSpeed) {

this.xSpeed = xSpeed;

}

// Getter for ySpeed

public float getYSpeed() {

return ySpeed;

}

// Setter for ySpeed

public void setYSpeed(float ySpeed) {

this.ySpeed = ySpeed;

}

// Sets both xSpeed and ySpeed

public void setSpeed(float xSpeed, float ySpeed) {

this.xSpeed = xSpeed;

this.ySpeed = ySpeed;

}

// Returns xSpeed and ySpeed as an array

public float[] getSpeed() {

return new float[]{xSpeed, ySpeed};

}

// Moves the point based on its speed

public MovablePoint move() {

setX(getX() + xSpeed); // Update x coordinate

setY(getY() + ySpeed); // Update y coordinate

return this;

}

// toString method

@Override

public String toString() {

return super.toString() + ", speed=(" + xSpeed + ", " + ySpeed + ")";

}

}

1. **Demo.java**

package Problem\_two;

/\*\*

\* Demonstrates the functionality of Point and MovablePoint classes.

\*/

public class Demo {

public static void main(String[] args) {

// Test Point class

Point p1 = new Point(2.5f, 3.5f);

System.out.println("Point p1: " + p1);

p1.setXY(5.0f, 7.0f);

System.out.println("Updated Point p1: " + p1);

// Test MovablePoint class

MovablePoint mp1 = new MovablePoint(1.0f, 2.0f, 0.5f, 1.5f);

System.out.println("MovablePoint mp1: " + mp1);

// Move the MovablePoint and display the new position

mp1.move();

System.out.println("After moving mp1: " + mp1);

// Update speed and move again

mp1.setSpeed(2.0f, 3.0f);

mp1.move();

System.out.println("After changing speed and moving mp1 again: " + mp1);

}

}

1. **Screenshot of the running code’s output**

**Point p1: (2.5, 3.5)**

**Updated Point p1: (5.0, 7.0)**

**MovablePoint mp1: (1.0, 2.0), speed=(0.5, 1.5)**

**After moving mp1: (1.5, 3.5), speed=(0.5, 1.5)**

**After changing speed and moving mp1: (3.5, 6.5), speed=(2.0, 3.0)**

**Problem#3:**

# Abstract Classes

Consider the following shapes; Ellipse, Circle, Triangle, EquilateralTriangle. Each shape should have a name, a method to compute its perimeter, and another method to compute its area. The name should be an instance variable of type String. Design your inheritance hierarchy with the common features in the **Abstract** superclass Shape. Notice that the area and perimeter are common to all Shapes, but we don’t know how to compute the area or perimeter for a general shape.

The ellipse class has a major and minor axes a and b, respectively. The constructor should assign the largest value to a and smallest to b. The area and perimeters of an ellipse are:

Perimeter = P = π [Note that if *a* = *b* = *r*, then P = 2π*r*]

Area = A = π*ab*

The Triangle class has three instance variables side1, side2, and side3. The formula for the area and perimeter of a general Triangle with sides A, B, and C is given by.





The condition for any three positive values to make sides of a Triangle is:

side1+side2>side3 and side2+side3>side1 and side3+side1>side2

You need to check this condition inside the constructor. If it is not satisfied, print an error message and terminate the program, otherwise make your Triangle object.

The three sides of the equilateral triangle are equal.

Make a Test class where you make objects from the different classes and store them in an array of type Shape. Then, make a loop and print the objects name, area, and perimeter through toString i.e. you need to override toString in the Shape class only.

**Deliverables:**

**Complete and working-class files with proper comments.**

1. **Shape.java**

package Problem\_three;

/\*\*

\* Abstract base class for all shapes.

\*/

public abstract class Shape {

protected String name; // Name of the shape

// Constructor

public Shape(String name) {

this.name = name;

}

// Abstract methods to compute area and perimeter

public abstract double getArea();

public abstract double getPerimeter();

// Override toString to return name, area, and perimeter

@Override

public String toString() {

return "Shape: " + name +

", Area: " + getArea() +

", Perimeter: " + getPerimeter();

}

}

1. **Circle.java**

package Problem\_three;

/\*\*

\* Represents a Circle (a special case of an Ellipse).

\*/

public class Circle extends Ellipse {

// Constructor

public Circle(double radius) {

super(radius, radius);

this.name = "Circle"; // Override name

}

}

1. **Ellipse.java**

package Problem\_three;

/\*\*

\* Represents an Ellipse shape.

\*/

public class Ellipse extends Shape {

private double a; // Major axis

private double b; // Minor axis

// Constructor

public Ellipse(double axis1, double axis2) {

super("Ellipse");

if (axis1 > axis2) {

this.a = axis1;

this.b = axis2;

} else {

this.a = axis2;

this.b = axis1;

}

}

// Compute area

@Override

public double getArea() {

return Math.PI \* a \* b;

}

// Compute perimeter (approximation)

@Override

public double getPerimeter() {

return Math.PI \* (3 \* (a + b) - Math.sqrt((3 \* a + b) \* (a + 3 \* b)));

}

}

1. **Triangle.java**

package Problem\_three;

/\*\*

\* Represents a Triangle shape.

\*/

public class Triangle extends Shape {

private double side1;

private double side2;

private double side3;

// Constructor

public Triangle(double side1, double side2, double side3) {

super("Triangle");

// Validate triangle inequality

if (side1 + side2 > side3 && side2 + side3 > side1 && side3 + side1 > side2) {

this.side1 = side1;

this.side2 = side2;

this.side3 = side3;

} else {

throw new IllegalArgumentException("Invalid sides for a triangle.");

}

}

// Compute area using Heron's formula

@Override

public double getArea() {

double s = getPerimeter() / 2.0; // Semi-perimeter

return Math.sqrt(s \* (s - side1) \* (s - side2) \* (s - side3));

}

// Compute perimeter

@Override

public double getPerimeter() {

return side1 + side2 + side3;

}

}

1. **EquilateralTriangle.java**

package Problem\_three;

/\*\*

\* Represents an Equilateral Triangle (a special case of a Triangle).

\*/

public class EquilateralTriangle extends Triangle {

// Constructor

public EquilateralTriangle(double side) {

super(side, side, side);

this.name = "Equilateral Triangle"; // Override name

}

}

1. **Demo.java**

package Problem\_three;

/\*\*

\* Demonstrates the functionality of Shape and its subclasses.

\*/

public class Demo {

public static void main(String[] args) {

// Create an array of shapes

Shape[] shapes = new Shape[4];

// Add shapes to the array

shapes[0] = new Circle(5.0);

shapes[1] = new Ellipse(7.0, 5.0);

shapes[2] = new Triangle(3.0, 4.0, 5.0);

shapes[3] = new EquilateralTriangle(6.0);

// Print details of each shape

for (Shape shape : shapes) {

System.out.println(shape);

}

}

}

1. **Screenshot of the running code’s output**

Shape: Circle, Area: 78.53981633974483, Perimeter: 31.41592653589793

Shape: Ellipse, Area: 109.95574287564276, Perimeter: 37.96136732875541

Shape: Triangle, Area: 6.0, Perimeter: 12.0

Shape: Equilateral Triangle, Area: 15.588457268119896, Perimeter: 18.0

**Problem#4:**

# Interfaces

Some OOP languages such as C++ allow a sub-class to inherit from more than one super class (multiple-inheritance). While this has some advantages, it makes such languages complex. To avoid such complexities, Java does not allow for multiple-inheritance. However, a lot of the advantages of multiple-inheritance can be achieved using **Interfaces**.

 An interface is similar to a class but with the following restrictions:

* All methods are implicitly **abstract** and **public**
* An interface cannot have instance variables. However, an Interface may have constants (final variables) and these are implicitly public and static. Also they are inherited by any class that implements the interface.
* An Interface can extend another interface and it is implemented by a class using the ***implements*** keyword. In fact, a class may implement any number of interfaces.

Consider an interface Scalable with a method scale of type void. It takes the scaling factor as a parameter. Make the shape class defined above implement the Scalable interface. Note that since Shape is abstract, it does not have to implement scale method.

Make the appropriate subclasses override scale method by multiplying their instance variables by the scale factor.

Modify the above Test class so that you add a static method that receives an array of Type Scalable, and a scale factor. This method should visit all the elements of the Scalable array and call the scale method with the scale factor passed to the static method. You should print your objects before and after scaling.

**Deliverables:**

**Complete and working-class files with proper comments.**

1. **Shape.java**

package Problem\_foure;

/\*\*

\* Abstract base class for all shapes implementing Scalable.

\*/

public abstract class Shape implements Scalable {

protected String name; // Name of the shape

// Constructor

public Shape(String name) {

this.name = name;

}

// Abstract methods to compute area and perimeter

public abstract double getArea();

public abstract double getPerimeter();

// Override toString to return name, area, and perimeter

@Override

public String toString() {

return "Shape: " + name +

", Area: " + getArea() +

", Perimeter: " + getPerimeter();

}

}

1. **Circle.java**

package Problem\_foure;

/\*\*

\* Represents a Circle (a special case of an Ellipse).

\*/

public class Circle extends Ellipse {

// Constructor

public Circle(double radius) {

super(radius, radius);

this.name = "Circle"; // Override name

}

// Scale the radius (uses Ellipse's scale method)

@Override

public void scale(double factor) {

super.scale(factor);

}

}

1. **Ellipse.java**

package Problem\_foure;

/\*\*

\* Represents an Ellipse shape.

\*/

public class Ellipse extends Shape {

private double a; // Major axis

private double b; // Minor axis

// Constructor

public Ellipse(double axis1, double axis2) {

super("Ellipse");

if (axis1 > axis2) {

this.a = axis1;

this.b = axis2;

} else {

this.a = axis2;

this.b = axis1;

}

}

// Compute area

@Override

public double getArea() {

return Math.PI \* a \* b;

}

// Compute perimeter (approximation)

@Override

public double getPerimeter() {

return Math.PI \* (3 \* (a + b) - Math.sqrt((3 \* a + b) \* (a + 3 \* b)));

}

// Scale the axes

@Override

public void scale(double factor) {

this.a \*= factor;

this.b \*= factor;

}

}

1. **Triangle.java**

package Problem\_foure;

/\*\*

\* Represents a Triangle shape.

\*/

public class Triangle extends Shape {

private double side1;

private double side2;

private double side3;

// Constructor

public Triangle(double side1, double side2, double side3) {

super("Triangle");

// Validate triangle inequality

if (side1 + side2 > side3 && side2 + side3 > side1 && side3 + side1 > side2) {

this.side1 = side1;

this.side2 = side2;

this.side3 = side3;

} else {

throw new IllegalArgumentException("Invalid sides for a triangle.");

}

}

// Compute area using Heron's formula

@Override

public double getArea() {

double s = getPerimeter() / 2.0; // Semi-perimeter

return Math.sqrt(s \* (s - side1) \* (s - side2) \* (s - side3));

}

// Compute perimeter

@Override

public double getPerimeter() {

return side1 + side2 + side3;

}

// Scale the sides

@Override

public void scale(double factor) {

this.side1 \*= factor;

this.side2 \*= factor;

this.side3 \*= factor;

}

}

1. **EquilateralTriangle.java**

package Problem\_foure;

/\*\*

\* Represents an Equilateral Triangle (a special case of a Triangle).

\*/

public class EquilateralTriangle extends Triangle {

// Constructor

public EquilateralTriangle(double side) {

super(side, side, side);

this.name = "Equilateral Triangle"; // Override name

}

// Scale the sides (uses Triangle's scale method)

@Override

public void scale(double factor) {

super.scale(factor);

}

}

1. **Scalable.java**

package Problem\_foure;

/\*\*

\* Interface for scalable shapes.

\*/

public interface Scalable {

// Abstract method to scale shapes

void scale(double factor);

}

1. **Demo.java**

package Problem\_foure;

/\*\*

\* Demonstrates the functionality of Shape and Scalable interface.

\*/

public class Demo {

// Static method to scale an array of Scalable objects

public static void scaleShapes(Scalable[] shapes, double factor) {

System.out.println("Before Scaling:");

for (Scalable shape : shapes) {

System.out.println(shape);

}

// Scale each shape

for (Scalable shape : shapes) {

shape.scale(factor);

}

System.out.println("\nAfter Scaling:");

for (Scalable shape : shapes) {

System.out.println(shape);

}

}

public static void main(String[] args) {

// Create an array of Scalable objects

Scalable[] shapes = new Scalable[4];

shapes[0] = new Circle(5.0);

shapes[1] = new Ellipse(7.0, 5.0);

shapes[2] = new Triangle(3.0, 4.0, 5.0);

shapes[3] = new EquilateralTriangle(6.0);

// Scale the shapes with a factor of 2

scaleShapes(shapes, 2.0);

}

}

1. **Screenshot of the running code’s output**

**Before Scaling:**

**Shape: Circle, Area: 78.53981633974483, Perimeter: 31.41592653589793**

**Shape: Ellipse, Area: 109.95574287564276, Perimeter: 37.96136732875541**

**Shape: Triangle, Area: 6.0, Perimeter: 12.0**

**Shape: Equilateral Triangle, Area: 15.588457268119896, Perimeter: 18.0**

**After Scaling:**

**Shape: Circle, Area: 314.1592653589793, Perimeter: 62.83185307179586**

**Shape: Ellipse, Area: 439.822971502571, Perimeter: 75.92273465751082**

**Shape: Triangle, Area: 24.0, Perimeter: 24.0**

**Shape: Equilateral Triangle, Area: 62.353829072479584, Perimeter: 36.0**

# Submission:

Please create a public github repository and upload all the java files, screen shots and feedback questions. Submit the link to that repository.