OBJECT-ORIENTED PROGRAMMING
LAB 7: POLYMORPHISM, ABSTRACTION

I. Objective

After completing this first lab tutorial, you can:

- Understand *polymorphism* and *abstraction* in OOP.

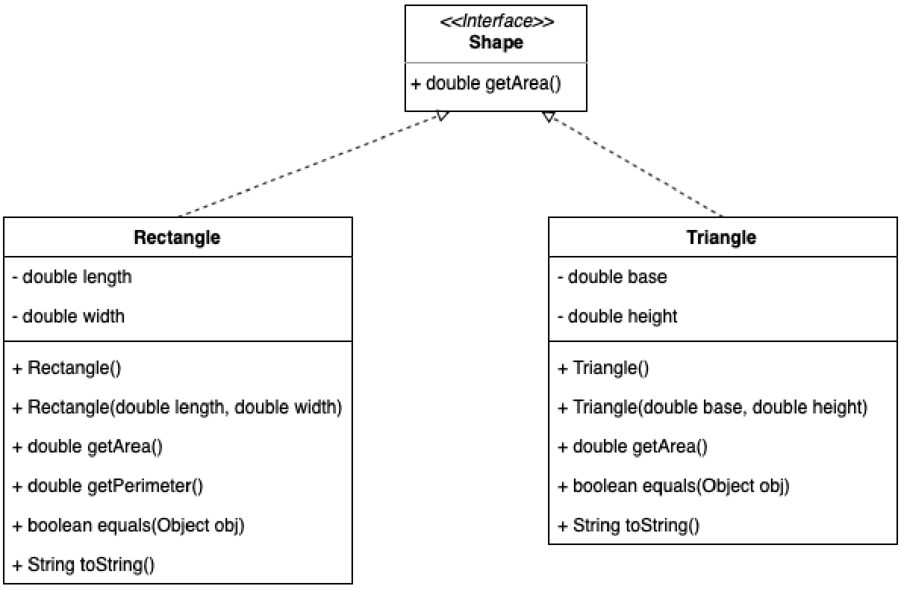
II. Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

In this section, we will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

In the following example, we have two real objects, which are Rectangle and Triangle, and a general object Shape. In reality, we don't need to create a Shape object, since it does not have any behavior. The question is how can we prevent users from creating Shape object.

We have two approaches, the first one is taking advantage of *interface* and the second is using *abstract* class.


<<Interface>>
Shape
+ double getArea()
Rectangle
- double length
- double width
+ Rectangle()
+ Rectangle(double length, double width)
+ double getArea()
+ double getPerimeter()
+ boolean equals(Object obj)
+ String toString()
Triangle
- double base
- double height
+ Triangle()
+ Triangle(double base, double height)
+ double getArea()
+ boolean equals(Object obj)
+ String toString()

```java
// Shape.java
public interface Shape {
    public double getArea();
}
```

```java
// Rectangle.java
public class Rectangle implements Shape {
    private double length;
    private double width;

    public Rectangle() {
        this.length = 0.0;
        this.width = 0.0;
    }

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    public double getLength() {
        return this.length;
    }

    public double getWidth() {
        return this.width;
    }

    public void setLength(double length) {
        this.length = length;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    @Override
    public double getArea() {
        return this.length * this.width;
    }

    public double getPerimeter() {
        return (this.length + this.width) * 2.0;
    }

    @Override
    public String toString() {
        reutrn "Rectangle{" + "length=" + this.length + ", width=" + this.width + '}';
    }
}
```

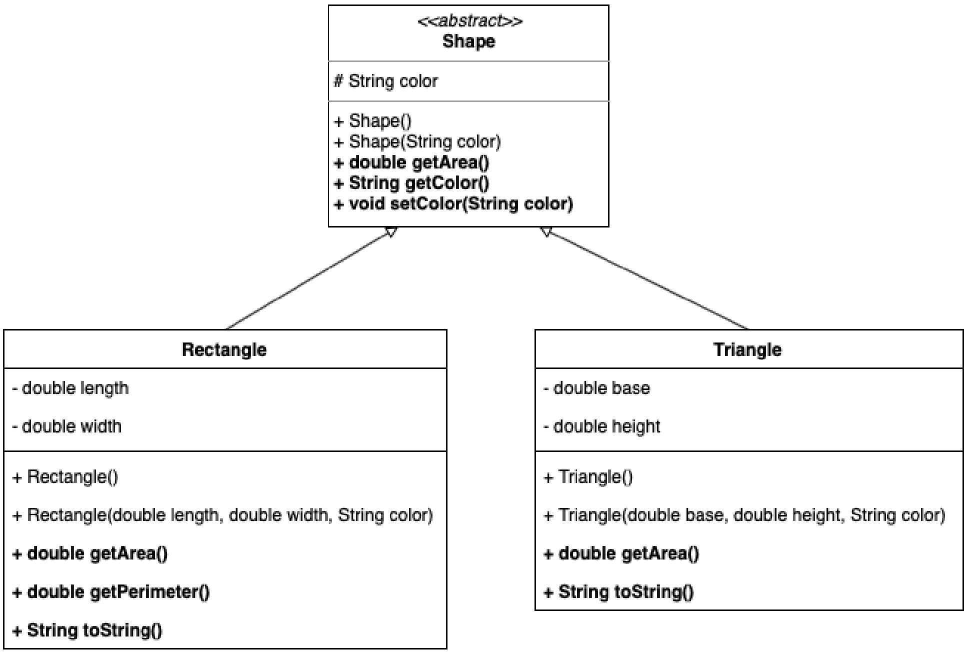
```
// Test.java
public class Test {
    public static void main(String[] args) {
        Shape s = new Rectangle(3, 4);
        System.out.println(s.toString());
        System.out.println("Area = " + s.getArea());

        s = new Triangle(4, 5);
        System.out.println(s.toString());
        System.out.println("Area = " + s.getArea());
    }
}
```

III. Abstraction

Abstraction is process of hiding the implementation details from the user, only the functionality will be provided to the user. For example, in email system, to send an email, user need only to provide recipient email, the content, and click send. All implementation of the system is hidden.

Java provides a mechanism to allow a program to achieve abstraction, by using abstract keyword. The abstract keyword can be used for class and method definition. For example, in the following diagram, we will define an abstract class, *Shape*, which contains *color* attribute and *getArea*() behavior. That means, every derived object from *Shape* will have *color* information and *getArea*() method.

You should notice that if you define an abstract class, the method must be either an abstract method or an implemented method. An abstract class cannot be instantiated. This usually happens because it has one or more abstract methods. It must be implemented in concrete (i.e., non-abstract) subclasses.


<<abstract>>
Shape
# String color
+ Shape()
+ Shape(String color)
+ double getArea()
+ String getColor()
+ void setColor(String color)
Rectangle
- double length
- double width
+ Rectangle()
+ Rectangle(double length, double width, String color)
+ double getArea()
+ double getPerimeter()
+ String toString()
Triangle
- double base
- double height
+ Triangle()
+ Triangle(double base, double height, String color)
+ double getArea()
+ String toString()

```java
// Shape.java
public abstract class Shape {
    protected String color;

    public Shape() {
        this.color = "";
    }

    public Shape(String color) {
        this.color = color;
    }

    public abstract double getArea()

    public String getColor() {
        return this.color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

```java
// Rectangle.java
public class Rectangle extends Shape {
    private double length;
    private double width;

    public Rectangle() {
        super();
        this.length = 0;
        this.width = 0;
    }

    public Rectangle(double length, double width, String color) {
        super(color);
        this.length = length;
        this.width = width;
    }

    @override
    public double getArea() {
        return this. length * this.width;
    }

    public double getPerimeter() {
        return (this.length + this.width) * 2.0;
    }

    public String toString() {
        return "Rectangle{" + "length=" + length +
               ", width=" + width +
               ", color=" + color + '}';
    }
}
```

```
// Test.java
public class Test {
    public static void main(String[] args) {
        Shape s = new Rectangle(3, 4, "white");
        System.out.println(s.toString());
        System.out.println("Area = " + s.getArea());

        s = new Triangle(4, 5, "black");
        System.out.println(e.toString());
        System.out.println("Area = " + s.getArea());
    }
}
```

IV. instanceof operator and equals method

Sometime, we need to compare two objects, but the object may have many attributes, Java doesn't know which criteria to evaluate two objects equal or not. So you need to override the equals method from Object class. Here is two examples:

a) You have two objects Rectangle and you said that two rectangle equal when their area are the same. So we have class Rectangle and override the equals method:

```
public class Rectangle{
    private double width;
    private double length;

    public Rectangle(double width, double length){
        this.width = width;
        this.length = length;
    }

    public double getArea(){
        return this.length * this.width;
    }

    @Override
    public boolean equals(Object obj){
        if(obj instanceof Rectangle){
            Rectangle temp = (Rectangle) obj;
            if(this.getArea() == temp.getArea()){
                return true;
            }
        }
        return false;
    }
}
```

You can test this Rectangle class as follows.

```
public class TestRectangle{
    public static void main(String[] args){
        Rectangle rec = new Rectangle(6, 8);
        Rectangle rec1  = new Rectangle(6, 8);
        Rectangle rec2 = new Rectangle(7,9);

        System.out.println(rec.equals(rec1));
        System.out.println(rec.equals(rec2));
    }
}
```

b) You have two object Fraction and two fraction equal when they have the numerators and the denominators equal in reduced form. We can define the Fraction class:

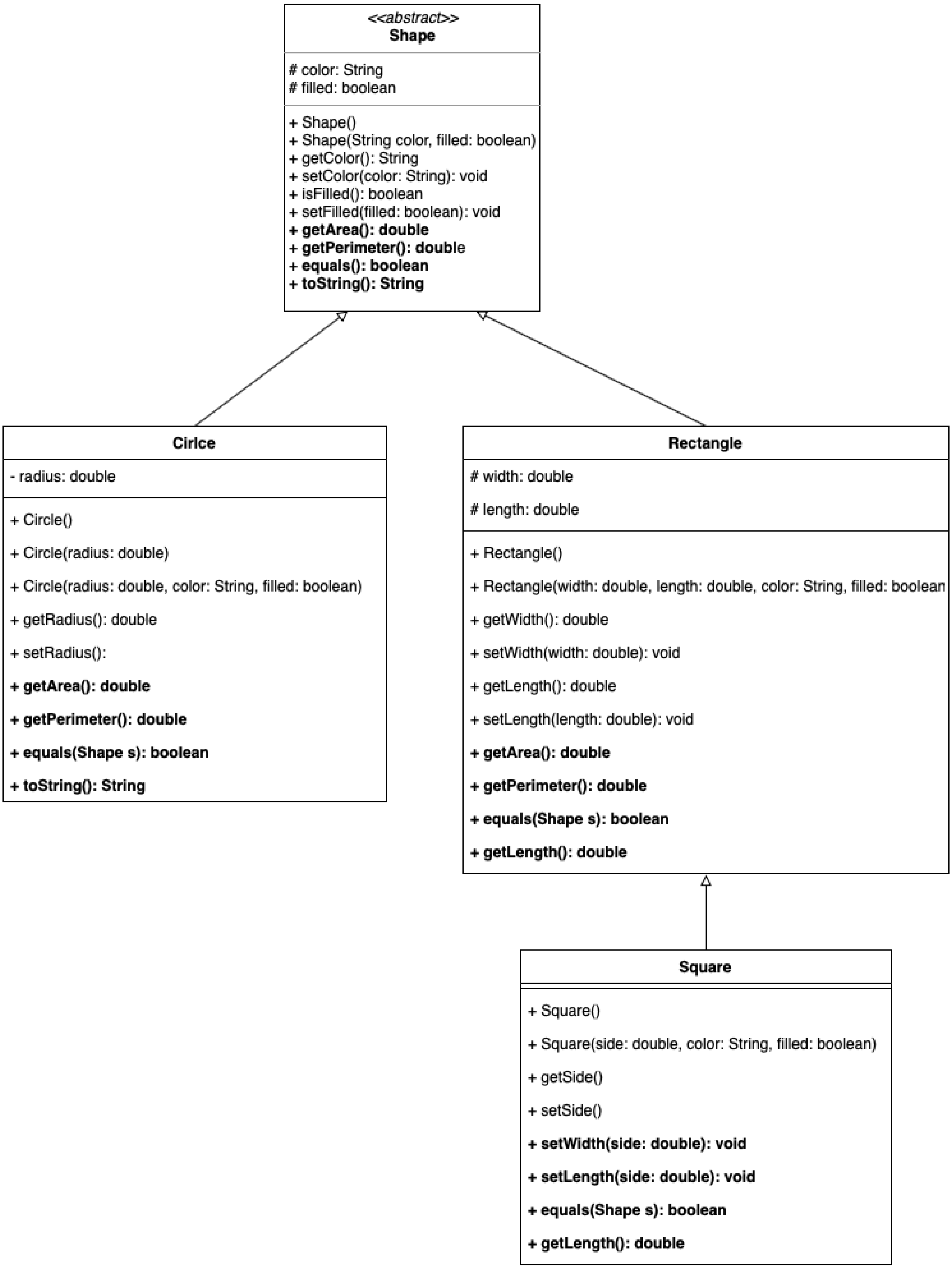
```
public class Fraction{
    private int numerator;
    private int denominator;

    public Fraction(int num, int den){
        this.numerator = num;
        this.denominator = den;
    }

    public Fraction reducer(){
        int gcd = this.gcd(this.numerator, this.denominator);
        return new Fraction(this.numerator/gcd, this.denominator/gcd);
    }

    private int gcd(int num, int den){
        while(num != den){
            if(num > den){
                num -= den;
            }else{
                den -= num;
            }
        }
        return num;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Fraction){
            Fraction temp = (Fraction) obj;
            temp = temp.reducer();
            Fraction temp1 = this.reducer();
            if(temp.numerator == temp1.numerator && temp.denominator == temp1.denominator){
                return true;
            }
        }
        return false;
    }
}
```

You can test this Fraction class as follows.

```
public class TestFraction{
    public static void main(String[] args){
        Fraction f = new Fraction(3, 4);
        Fraction f1 = new Fraction(6, 8);
        Fraction f2 = new Fraction(3, 2);

        System.out.println(f.equals(f1));
        System.out.println(f.equals(f2));
    }
}
```

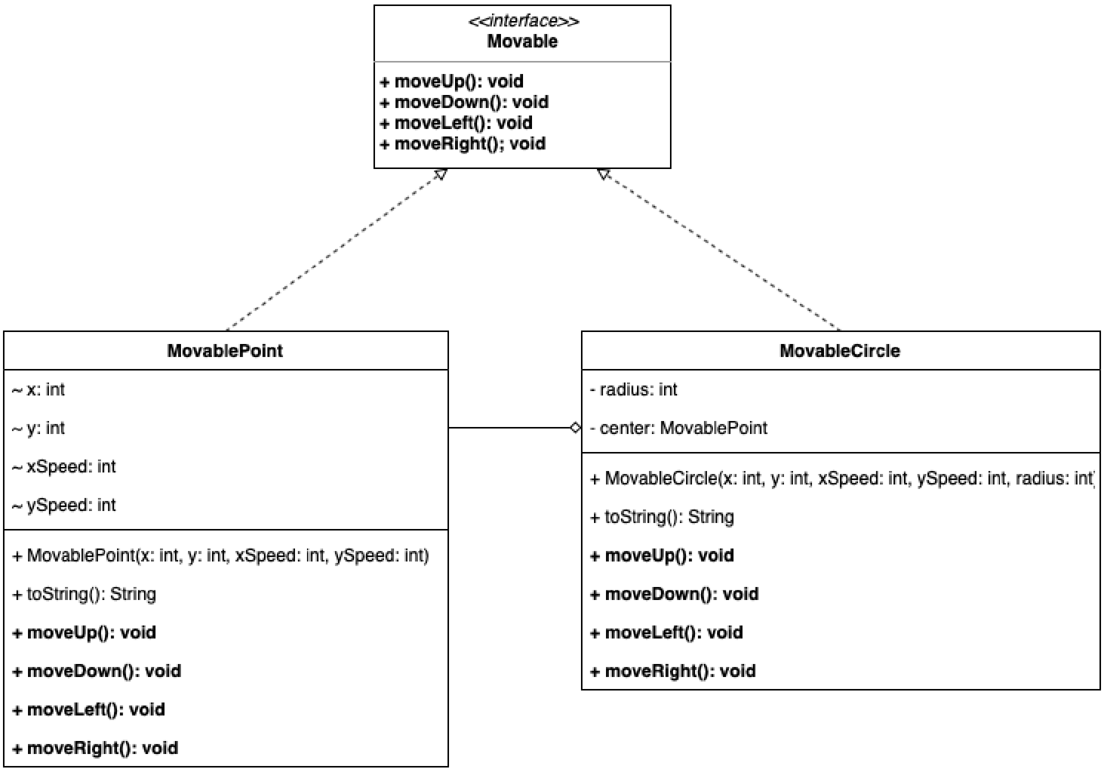
V. Classwork

1. Continue the above examples, implement the Triangle class.
2. Abstract superclass Shape and its concrete subclasses.

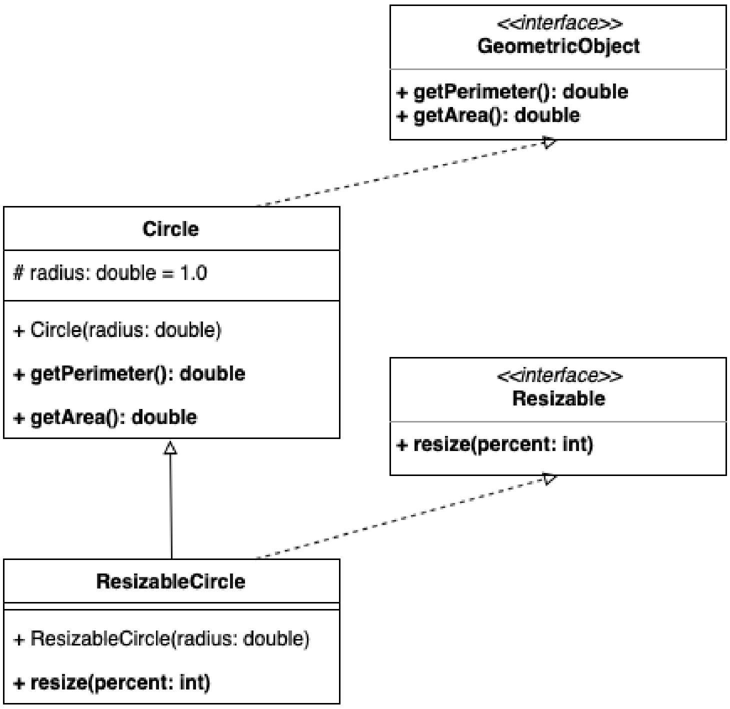

<<abstract>>
Shape
# color: String
# filled: boolean
+ Shape()
+ Shape(String color, filled: boolean)
+ getColor(): String
+ setColor(color: String): void
+ isFilled(): boolean
+ setFilled(filled: boolean): void
+ getArea(): double
+ getPerimeter(): double
+ equals(): boolean
+ toString(): String
Cirlce
- radius: double
+ Circle()
+ Circle(radius: double)
+ Circle(radius: double, color: String, filled: boolean)
+ getRadius(): double
+ setRadius():
+ getArea(): double
+ getPerimeter(): double
+ equals(Shape s): boolean
+ toString(): String
Rectangle
# width: double
# length: double
+ Rectangle()
+ Rectangle(width: double, length: double, color: String, filled: boolean
+ getWidth(): double
+ setWidth(width: double): void
+ getLength(): double
+ setLength(length: double): void
+ getArea(): double
+ getPerimeter(): double
+ equals(Shape s): boolean
+ getLength(): double
Square
+ Square()
+ Square(side: double, color: String, filled: boolean)
+ getSide()
+ setSide()
+ setWidth(side: double): void
+ setLength(side: double): void
+ equals(Shape s): boolean
+ getLength(): double

VI. Homework

1. Interface Movable and its implementations MovablePoint and MovableCircle.


<<interface>>
Movable
+ moveUp(): void
+ moveDown(): void
+ moveLeft(): void
+ moveRight(); void
MovablePoint
~ x: int
~ y: int
~ xSpeed: int
~ ySpeed: int
+ MovablePoint(x: int, y: int, xSpeed: int, ySpeed: int)
+ toString(): String
+ moveUp(): void
+ moveDown(): void
+ moveLeft(): void
+ moveRight(): void
MovableCircle
- radius: int
- center: MovablePoint
+ MovableCircle(x: int, y: int, xSpeed: int, ySpeed: int, radius: int
+ toString(): String
+ moveUp(): void
+ moveDown(): void
+ moveLeft(): void
+ moveRight(): void

2. Interface GeometricObject and Resizable.


<<interface>>
GeometricObject
+ getPerimeter(): double
+ getArea(): double
Circle
# radius: double = 1.0
+ Circle(radius: double)
+ getPerimeter(): double
+ getArea(): double
<<interface>>
Resizable
+ resize(percent: int)
ResizableCircle
+ ResizableCircle(radius: double)
+ resize(percent: int)