OBJECT-ORIENTED PROGRAMMING
LAB 7: POLYMORPHISM, ABSTRACTION

I. Objective

After completing this first lab tutorial, you can:

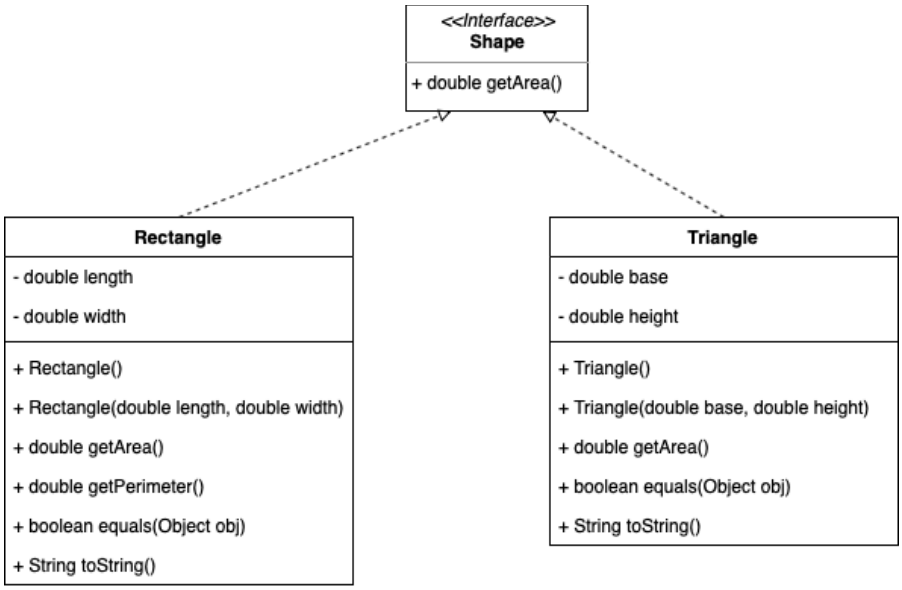
- Understand *polymorphism* and *abstraction* in OOP.

II. Polymorphism

Polymorphism is the behavior of functionality changes according to the actual type of data. We have two mechanisms are Overloading and Overriding to approach the polymorphism in Java. 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 through the example of the *interface* in Java.

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 objects.




```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() {
        return "Rectangle{" + "length=" + this.length + ", width=" +
            this.width + '}';
    }
}
```

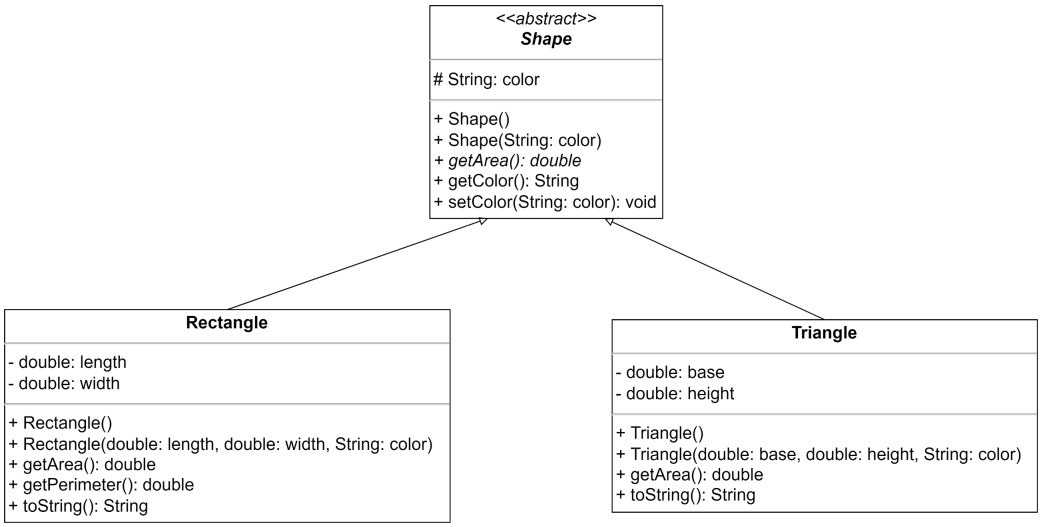
```java
// Test.java
public class Test {
    public static void main(String[] args) {
        Shape s = new Rectangle(4, 3);
        System.out.println(s.toString());
        System.out.println("Area = " + s.getArea()); // 12.0

        s = new Triangle(8, 7);
        System.out.println(s.toString());
        System.out.println("Area = " + s.getArea()); // 28.0
    }
}
```

III. Abstraction

Abstraction is the process of hiding the implementation details from the user, only the functionality will be provided to the user. For example, in the email system, to send an email, the user needs only to provide the recipient's email, and the email's content and click send. All implementation of the system is hidden. We have two approaches, the first one is taking advantage of the *interface* and the second is using *abstract* class.

Java provides a mechanism to allow a program to achieve abstraction using *abstract* keywords. The *abstract* keyword can be used for class and method definitions. For example, the following diagram 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 the *getArea*() method.

You should notice that if you define an abstract class, the method must be either abstract or implemented. 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.



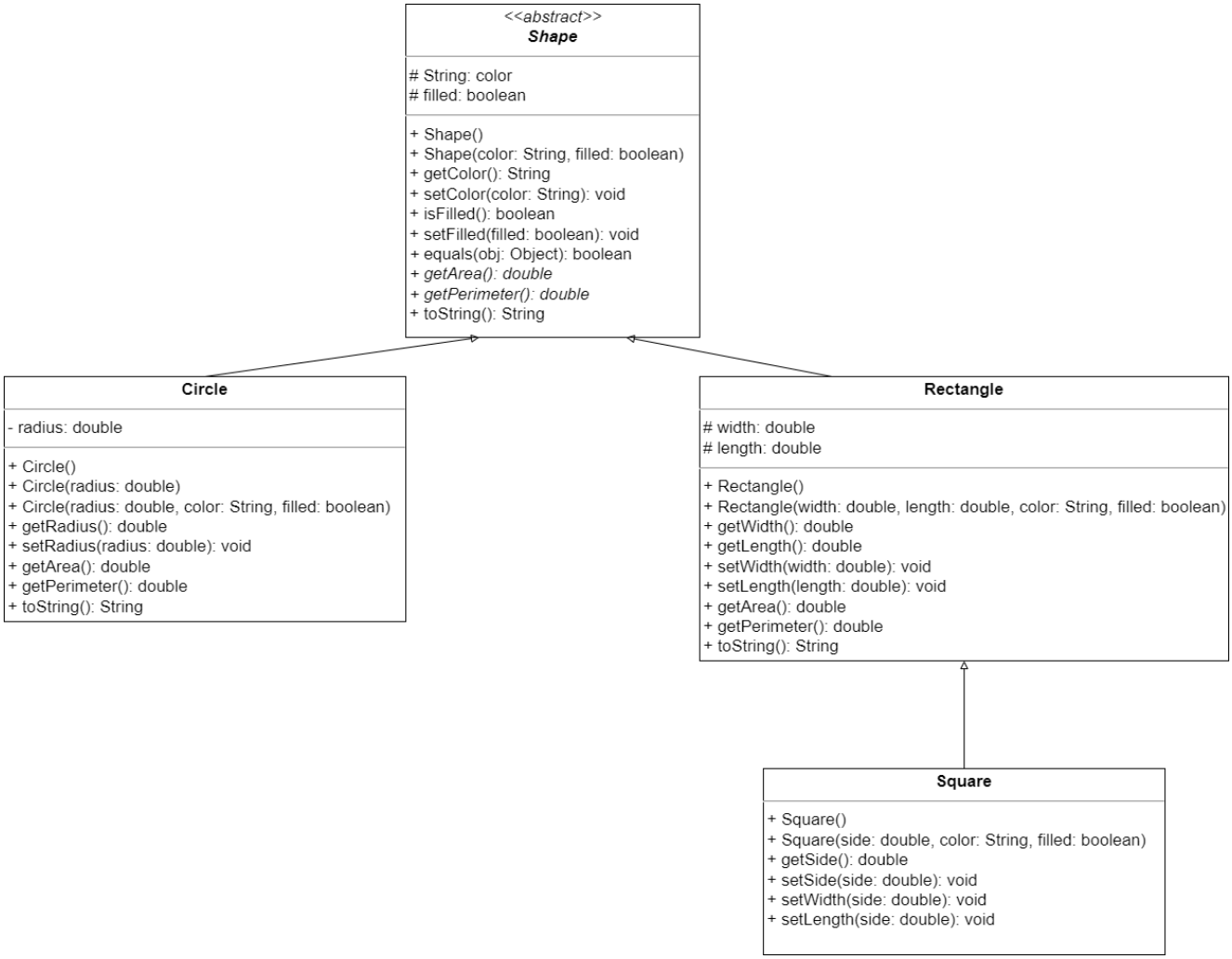
```java
// Shape.java
public abstract class Shape {
    protected String color;

    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 + '}';
    }
}
```

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

        s = new Triangle(8, 7, "black");
        System.out.println(e.toString());
        System.out.println("Area = " + s.getArea());
    }
}
```

IV. Exercises

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



All Shapes will compare based on the area, if two Shapes have the same area, these Shapes will be equal.

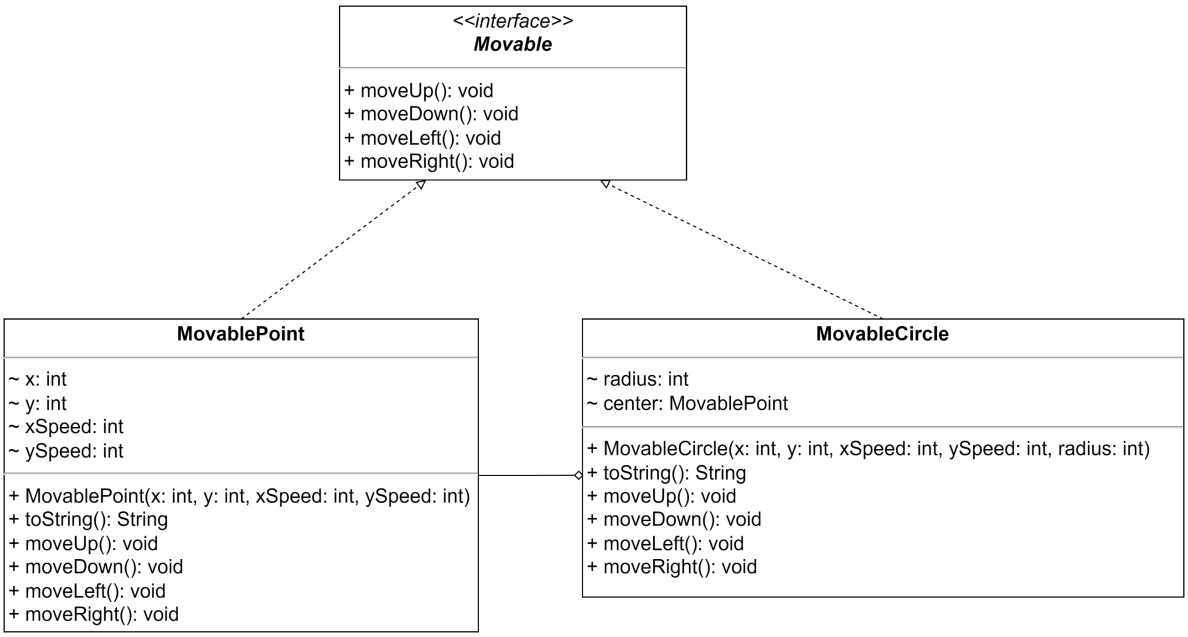
Write a class with a *main* method to test your work.

Assume that in the *main* method of the test class, you have a list containing any shapes. Your work is to find the shape that has the largest area in this list.

Example: Find the shape that has the largest area in the list below.

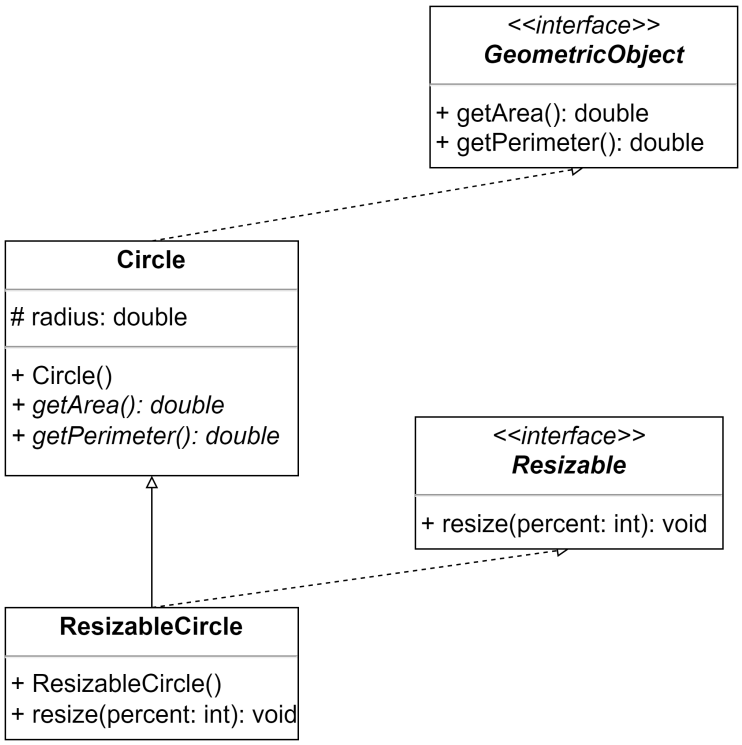
```
Shape[] shapes = new Shape[5];
shapes[0] = new Circle(4, "Red", true);
shapes[1] = new Rectangle(8, 4, "Blue", true);
shapes[2] = new Square(10, "Black", true);
shapes[3] = new Circle(9);
shapes[4] = new Rectangle(12, 8, "Blue", true);
```

3. Implement the below interface and classes to manage points in Descartes's coordinate system. Given that, the *move* methods will change the coordinate *x* or *y* based on the *xSpeed* or *ySpeed*.



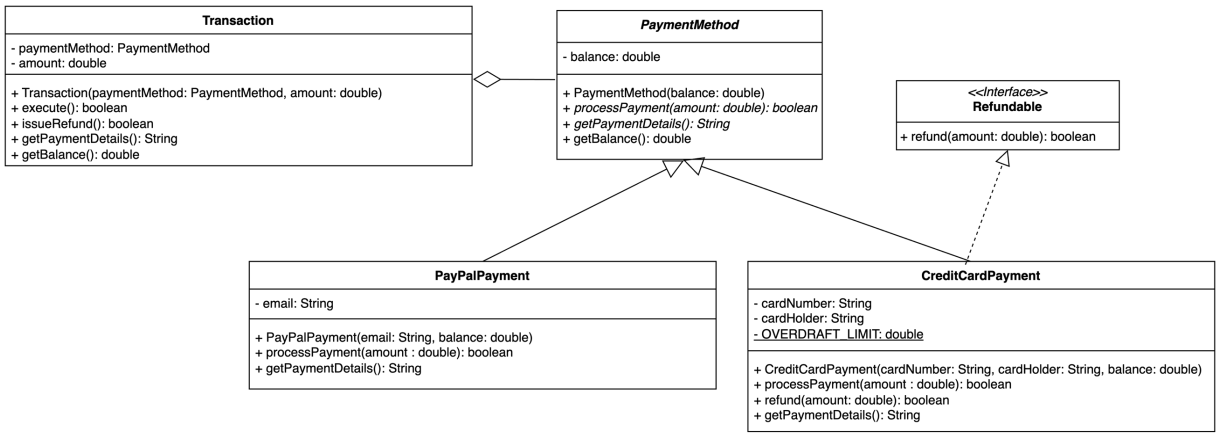
Write a class with a *main* method to test your work.

4. Implement the below interface and classes. Know that the *resize()* method will change the *radius* based on the *percent* parameter.



Write a class with a *main* method to test your work.

5. **Exercise: Payment Processing System**



In this exercise, you will create a basic payment processing system that supports multiple payment methods and demonstrates core object-oriented programming principles such as inheritance, encapsulation, and polymorphism. The objective is to model a flexible system where different types of payment methods can be handled uniformly.

Your system will include an abstract class called **PaymentMethod**, representing a generic payment method. This class should maintain a balance attribute and provide methods for checking the current *balance*. Additionally, it will declare two abstract methods: one for

processing payments and another for retrieving payment details. By defining these methods as abstract, you will enforce that all subclasses must implement their own version of these functionalities.

You will also define an interface named **Refundable** to represent payment methods that support refunds. This interface will declare a single method for processing refunds. Not all payment methods will implement this interface, so your system should be able to handle cases where a refund is not supported.

To apply these concepts, you will implement two specific payment methods: **CreditCardPayment** and **PayPalPayment**.

The **CreditCardPayment** class should include attributes for the *cardNumber*, *cardHolder*, and *balance*. It should also allow spending beyond the available balance up to a fixed *OVERDRAFT_LIMIT*. Additionally, it should support refunds by implementing the **Refundable** interface.

The **PayPalPayment** class will be simpler, allowing payments only within the available *balance* and not supporting refunds. It should have an attribute representing the user's *email*.

Furthermore, you will implement a **Transaction** class to represent a single transaction between a customer and a payment method. This class should accept a *paymentMethod* and an *amount* to be processed. It should also provide the ability to issue refunds if the payment method supports it.

- The ***execute()*** method in the **Transaction** class is responsible for processing a transaction by attempting to deduct the specified amount from the associated **PaymentMethod**. This method should return a boolean value indicating whether the transaction was successful or not. It will check the balance and overdraft limits (if applicable) before completing the transaction.
- The ***issueRefund()*** method in the **Transaction** class is responsible for processing a refund. This method checks if the associated **PaymentMethod** implements the **Refundable** interface. If it does, the refund is processed, and the balance is adjusted accordingly. Otherwise, the refund is considered unsupported. The method should return a boolean value indicating whether the refund was successful or not.

Finally, create a simple **PaymentProcessor** class to simulate real-world usage. This class should contain **main** method and instantiate various payment methods, process payments, attempt refunds where applicable, and display the resulting *balance* for each method.

--- THE END ---