Objectives: In this lab, the following topics will be covered

1. Abstract Classes and Polymorphism
2. Interfaces
3. Examples & Exercises for practice

1. Abstract Classes and Polymorphism:

An abstract class usually defines a concept. Abstract classes have no implementation and they are only used as generic superclasses. An abstract class may contain:

- An abstract method (no implementation)
- A non-abstract method
- Instance variables

Note that it is not necessary for an abstract class to have an abstract method. An abstract class is always preceded by the keyword abstract. As an example, we provide the following hierarchy of an abstract class Shape and its non-abstract children:

In this lab we’ll walk through the following class hierarchy:

Shape.java

Note the 2 abstract methods in the following abstract class. Also note the presence of two non-abstract methods name() and toString().
Circle.java
The first non-abstract child of Shape is Circle. Note that all abstract methods have been provided with an implementation.

```java
public class Circle extends Shape {
    private double radius;
    public Circle(double r) {
        radius = r;
    }
    public double area() {
        return Math.PI * (radius * radius);
    }
    public double perimeter() {
        return 2.0 * Math.PI * radius;
    }
    public double getRadius() {
        return radius;
    }
}
```

Rectangle.java
This class is also derived from the Shape class.

```java
public class Rectangle extends Shape {
    private double length;
    private double width;
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    public double area() {
        return length * width;
    }
    public double perimeter() {
        return 2 * (length + width);
    }
    public double getLength() {
        return length;
    }
    public double getWidth() {
        return width;
    }
}
```

Square.java

```java
public class Square extends Rectangle {
    public Square(double length) {
        super(length, length);
    }
}
```
TestShapes.java:
This class basically creates several Shape objects and prints them. Note the up-casting in the assignment of the array randomShapes. Since randomShapes is of type Shape, any sub-class of Shape can be assigned to it. Such a reference is called a polymorphic reference. Also note the polymorphic call to the area() and perimeter() methods when the toString() method of each one of the classes is called depending upon the actual shape.

The assignments here are done randomly to illustrate the point that the methods executed belong to the actual object which is dynamically bound to the object. Consider the line highlighted at the end: System.out.println(randomShapes[i]); Looking at the code, you cannot tell which toString() method will be called as the assignment of objects, the length of the array and even the dimensions of the shapes are randomly assigned only to be decided at runtime.

```java
import java.util.Random;
public class TestShapes {
    public static Shape[] createShape() {
        final int SIZE = 5;
        final double DIMENSION = 100;
        final int NUMBEROFSHAPES = 3;

        Random generator = new Random();

        // create an array having b/w 1 and SIZE entries
        Shape[] randomShapes = new Shape[generator.nextInt(SIZE) + 1];

        for(int i = 0; i < randomShapes.length; i++)
        {
            // randomly generate values b/w 0 and NUMBEROFSHAPES - 1
            int assigner = generator.nextInt(NUMBEROFSHAPES);

            switch(assigner) {
                case 0: randomShapes[i] = new Rectangle(generator.nextDouble()*DIMENSION,
                                                           generator.nextDouble()*DIMENSION);
                break;
                case 1: randomShapes[i] = new Circle(generator.nextDouble()*DIMENSION);
                break;
                case 2: randomShapes[i] = new Square(generator.nextDouble()*DIMENSION);
                break;
            }
        }
        return randomShapes;
    }

    public static void main(String[] args) {
        Shape[] randomShapes = TestShapes.createShape();
        for(int i = 0; i < randomShapes.length; i++)
            System.out.println(randomShapes[i]);
    }
}
```
DownCasting:
Consider the following program. Here each shape object is being tested whether it’s a Circle, Rectangle or a Square using the instanceof operator. According to that, it’s radius/length/width is also being printed by downcasting it to the required object.

DownCastingShapes.java:
```java
public class DownCastingShapes{
    public static void main(String[] args){
        Shape[] randomShapes = TestShapes.createShape();
        for(int i = 0; i < randomShapes.length; i++){
            System.out.println(randomShapes[i]);
            if(randomShapes[i] instanceof Circle)
                System.out.println("Radius = " + ((Circle) randomShapes[i]).getRadius());
            else if(randomShapes[i] instanceof Square)
                System.out.println("Length = " + ((Square) randomShapes[i]).getLength());
            else if(randomShapes[i] instanceof Rectangle)
                System.out.println("Length = " + ((Rectangle) randomShapes[i]).getLength() + "\nWidth = " + ((Rectangle) randomShapes[i]).getWidth());
        }
    }
}
```

Exercise:
1. Design and implement a subclass “EquilateralTriangle” having a double variable side denoting the three sides of the equilateral triangle [Note that since all the 3 sides are equal, the constructor will have only one parameter]. The area and perimeter of the equilateral triangle are given as follows:

Area = \( \frac{\sqrt{3}}{4} \times (side)^2 \)

Perimeter = 3*side

Provide accessor methods for the sides. Test your class using the TestShapes and DownCastingShapes classes.
2. Interfaces:

Interfaces in Java are a solution to the multiple-inheritance problem. In instances where a class exhibits a behavior of more than one class, interfaces are used.

Interfaces have only abstract methods and final (constant) variables. All methods in an interface are by default public and abstract.

Below we consider an extension to our Shape class. Here we discuss about elliptical shapes which have properties in addition to area and perimeter which are of interest to us.

Consider the Shape called Ellipse. An ellipse is basically nothing but an elongated circle. A circle has a radius. An ellipse has two measurements: the major axis \((a)\) and the minor axis \((b)\) with \(a \geq b\).

![Ellipse Diagram]

Note that in an ellipse if \(a = b\) then it becomes a circle.

The measure of “roundness” of an ellipse is given by its eccentricity \(e\) where the value of \(e\) is always between 0 and 1. For a circle \(e = 0\). The higher the value of \(e\), the higher is the deviation of the ellipse from its roundness.

Eccentricity is associated with a lot of other shapes as well e.g. hyperbola, parabola, etc. The various formulae for the ellipse are:

\[
\text{Perimeter} = P = \pi \sqrt{2(a^2 + b^2) - (a - b)^2}/2 \quad \text{[Note that if} \ a = b = r, \ \text{then} \ P = 2\pi r]\n\]

\[
\text{Area} = A = \pi ab
\]

\[
\text{Eccentricity} = e = \sqrt{1 - \frac{b^2}{a^2}}
\]

We can define an interface called Eccentric for all eccentric Shapes.

**Eccentric.java:**

```java
interface Eccentric {
    double eccentricity();
}
```
Note here that no access modifiers have been given for the method eccentricity. The reason is that all methods in an interface are by default public and abstract. Now we can define our class `Ellipse`.

**Ellipse.java:**

```java
public class Ellipse extends Shape implements Eccentric {
    double a, b;

    public Ellipse(double s1, double s2)
    {
        if (s1 < s2)
        {
            a = s2;
            b = s1;
        } else
        {
            a = s1;
            b = s2;
        }
    }

    public double perimeter()
    {
        // method body missing
    }

    public double area()
    {
        // method body missing
    }

    public double eccentricity()
    {
        // method body missing
    }

    public String toString()
    {
        // method body missing
    }
}
```

**Exercise:**

2. By looking at the formulae for an ellipse, provide the missing code for all of the methods in the class `Ellipse` including the `toString()` method. Test your program using the `TestShapes.java` class. Your output should look as follows (for an ellipse with \(a = 10\) and \(b = 7\)) (values are randomly generated).

```
Square
Area=100.0
Perimeter=40.0

Ellipse
Area=219.9114857512855
Perimeter=53.8212680240788
Eccentricity=0.714142842854285
Press any key to continue...
```
3. How about the following class `Circle`. Since a `Circle` is a special case of an `Ellipse`, will the output of `TestShapes.java` be affected if the following class is used instead of the class `Circle` used previously:

```java
public class Circle extends Ellipse{
    public Circle(double radius){
        super(radius, radius);
    }
}
```

With this modification, the class diagram would look as follows: