Exercises 06

1. Take the Point and Segment classes designed and implemented in previous weeks. Override the toString() and equals(Object obj) methods of these classes.
    a. **Note**: Previously we have not talked about overriding them.

2. Create a **Rectangle** class and override the *equals()* method of the **Object** class such that, if the width and height of the provided rectangle are the same as the current object, then return true.

```
class Rectangle {
    int width, height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    public boolean equals(Object obj) {
        Rectangle rect = (Rectangle) obj;
        // your code here
    }
}
```

    a. Test your code with the following statements:

```
Rectangle r1 = new Rectangle(5,10);
Rectangle r2 = new Rectangle(15,10);
Rectangle r3 = new Rectangle(5,10);

System.out.println(r1.equals(r2));
System.out.println(r1.equals(r3));
```

3. Now create a **Square** class that extends **Rectangle** and has a constructor with one argument – side. Test your code as given in the following example to see the power of OOP!

```
Object o1 = new Rectangle(5,10);
Object o2 = new Rectangle(15,15);
Object o3 = new Square(15);

System.out.println("Objects are identical: " + o1.equals(o2));
System.out.println("Objects are identical: " + o1.equals(o3));
System.out.println("Objects are identical: " + o2.equals(o3));
```

4. Override clone() method in Rectangle class and test your code by trying to clone a Rectangle object.
    a. Explain why clone() method has **protected** visibility in Object class.

    b. Discuss benefits of keeping it **protected** in extending class? When you must use **public** instead of **protected**?

5. Override clone() method for Point and Segment classes. Make sure you use most appropriate acces modifier for your solution.
    a. Do you think the way you implemented clone() method is deep or shallow one?
    b. Can you try the other one? Discuss the applications of each.
    c. **Note**: cloning objects can be achieved by in-memory serialization as well. We are not going to discuss it here.

6. Interfaces
    a. Define an interface named **My2DInt** which has two methods:
        i. double getArea()
        ii. double getPerimeter()
    b. Define an interface named **My3DInt** which has two methods:
        i. double getSurfaceArea()
        ii. double getVolume()
    c. Since Rectangle is a 2D type, it should implement My2DInt interface.
        i. We also had Square extending Rectangle.
            1. Do you think it will also have the same behavior or must be implemented?
    d. Define a class Cuboid which is a 3D version of a Rectangle. It will implement My3DInt though.
        i. Implement the unimplemented methods.
            1. Do you think Cuboid has the behavior declared in My2Dint as well?
    e. Test newly added methods.

7. Test BigInteger class. See that it can store values larger than the max value we can store in long primitive DT.
    a. Factorial
    b. Power

8. Test BigDecimal class. See the examples provided in the lecture. Check if they provide expected results or not.
    a. 2.35 – 1.95
    b. 1000000.0f + 1.2f - 1000000.0f
        i. See: https://www.h-schmidt.net/FloatConverter/IEEE754.html

9. [Bonus] Arithmetic Operations
    a. Examine the following code. Test it. Try to understand how it works!

```
public class Operand implements EvalInterface {

    private double value;
    private String label;

    public Operand(String label, double value) {
        this.label = label;
        this.value = value;
    }

    @Override
    public double toValue() {
        return value;
    }

    @Override
    public String toString() {
        return label;
    }

}
```

```
public interface EvalInterface {
    double toValue();

    String toString();
}
```

```
public abstract class BinaryOperation implements EvalInterface {
    private EvalInterface op1;
    private EvalInterface op2;
    private String label;

    public BinaryOperation(String label, EvalInterface op1, EvalInterface op2) {
        this.op1 = op1;
        this.op2 = op2;
        this.label = label;
    }

    protected abstract double calculate(EvalInterface op1, EvalInterface op2);

    @Override
    public double toValue() {
        return calculate(op1, op2);
    }

    @Override
    public String toString() {
        return "(" + op1.toString() + " " + label + " " + op2 + ")";
    }
}
```

```java
public class Sum extends BinaryOperation {

    public Sum(EvalInterface op1, EvalInterface op2) {
        super("+", op1, op2);
    }

    @Override
    protected double calculate(EvalInterface op1, EvalInterface op2) {
        return op1.toValue() + op2.toValue();
    }

}
```

```java
public class TestArithmeticOperations {
    Run | Debug
    public static void main(String[] args) {

        Operand x = new Operand("x", 5);
        Operand y = new Operand("y", 15);
        Operand z = new Operand("z", 3);

        Sum s = new Sum(x, y);

        Sum s2 = new Sum(new Sum(x, y), x);

        System.out.println(s.toString());
        System.out.println(s.toValue());

        System.out.println(s2.toString());
        System.out.println(s2.toValue());
    }
}
```

b. Once you understand what actually is happening, introduce some other BinaryOperations and test the new ones in TestArithmeticOperations.
    i. class Subtr
    ii. class Mult
    iii. class Div
c. Can you introduce a new **unary** operation as well? For that, you might want to have another abstract class **UnaryOperation**.
d. Once done, define some new unary operators and test the new ones in TestArithmeticOperations.
    i. class SuareRoot
    ii. class Factorial