Lab 03 – Programming with Objects and Classes

Objectives:

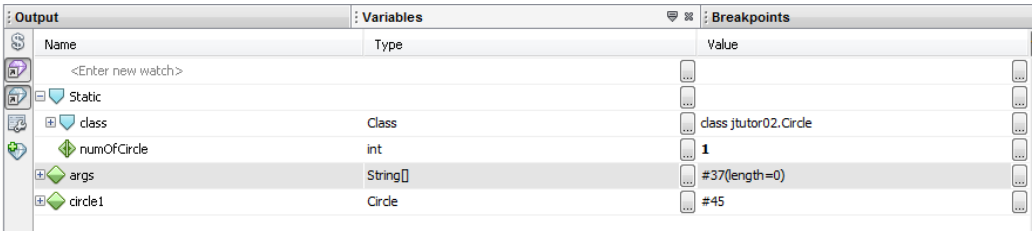
- Understand the use of constructors
- Learn the difference between static and non-static context
- Apply data encapsulation and information hiding principles in class design

1. The following methods are designed to exchange the values of their two input parameters. Analyse the code and indicate any of them should work as expected. Then, write a main method to test the two methods and prove your analysis. Discuss your observation with your classmates.

```
public class TestSwap {

    /**
     * Pass by value
     */
    public static void swap1(int n1, int n2) {
        int temp = n1;
        n1 = n2;
        n2 = temp;
    }

    /**
     * Pass by reference
     */
    public static void swap2(Integer n3, Integer n4) {
        Integer temp = n3;
        n3 = n4;
        n4 = temp;
    }

}
```

2. Create the following class ***Circle***, and set breakpoints in the three lines that highlighted in red color. Run the program in debug mode and step through the program line by line. Observe the changes of the static variable ***numOfCircle*** as well as the three circle objects.

```
public class Circle {

    double radius = 0;
    static int numOfCircle = 0;

    Circle(int radius) {
        this.radius = radius;
        Circle.numOfCircle++;
    }

    double getArea() {
        return radius * radius * Math.PI;
    }

    public static void main(String[] args) {
        Circle circle1 = new Circle(1);
        Circle circle2 = new Circle(2);
        Circle circle3 = new Circle(3);
        System.out.printf("Total %d circles created.\n", Circle.numOfCircle);
    }
}
```



The debug window indicates one circle object is just created

3. With the class *Circle* in Question 2 above, what will be displayed if the following code is compiled and executed? Indicate the problems and suggest amendments, if any.

```
public class CreateArrayOfCircle {

    public static void main(String[] args) {
        Circle[] circles = new Circle[3];
        for(int i=0; i<3; i++) {
            double area = circles[i].getArea();
            System.out.println(area);
        }
    }
}
```

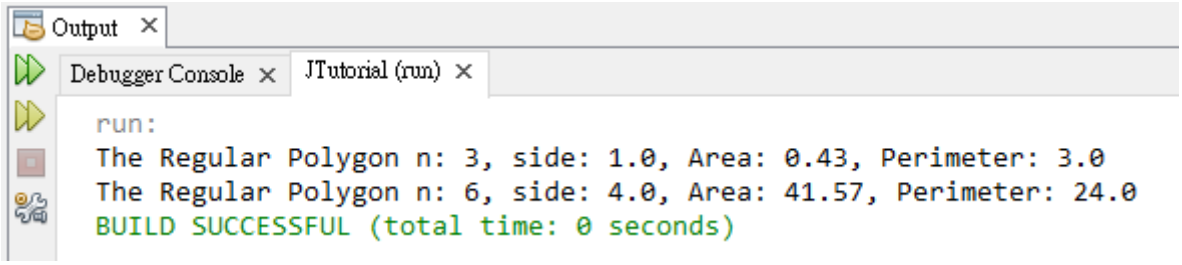
4. In an *n*-sided regular polygon, all sides have the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). Implement the class ***RegularPolygon*** with the following members:
   - a private int data field named ***n*** that defines the number of sides in the polygon with default values 3;
   - a private double data field named ***side*** that stores the length of the side with default value 1;
   - a no-arg constructor that creates a regular polygon with default values;
   - a constructor that creates a regular polygon with a specified number of sides and length of side;
   - accessor (getter) and mutator (setter) methods for all data fields, where the setters should assure that the side length must be >= 0 and the number of edges must be >= 3;
   - a method ***getPerimeter()*** that returns the perimeter of the polygon;
   - a method ***getArea()*** that returns the area of the polygon;

     The formula for computing the area of a regular polygon is:

$$Area = \frac{n \times s^2}{4 \times \tan\left(\frac{\Pi}{n}\right)}$$

     where *s* is the side length.

   - a ***toString()*** method that returns a string representation of a ***RegularPolygon*** object as shown below.



Write a test program, ***TestRegularPolygon***, that creates two ***RegularPolygon*** objects, a default polygon with the no-arg constructor, and another polygon with *n=6* and *side=4*. For each object, display its area and perimeter by invoking its toString() method as shown above.

Discuss how data encapsulation is applied in the design of ***RegularPolygon*** to protect its data integrity.

5. Design a class ***Matrix*** for representing 2D integer matrix. Apply the OO principles you have learnt in the course to your class design. The class should provide the following methods for a matrix object:
   - one or more constructors to initialize the matrix numbers using a 2D integer array;
   - a getter and a setter to get and set a particular number on the matrix (indicated by row and column no.);
   - a ***toString*** method to return a printable string representation of the matrix (see the sample output below);
   - an ***add*** method for adding another matrix to itself;
   - and a ***multiply*** method to return the product of two matrices.

   See the driver class below on how the Matrix class is supposed to be used.

```
// This serves as a test driver to test your Matrix class implementation.
// Run this test against your Matrix class when you finish your design.

public class TestMatrix {

    public static void main(String[] args) {
        // initialize both matrices
        Matrix m1 = new Matrix(new int[][]{{1,2},{3,4}});
        Matrix m2 = new Matrix(new int[][]{{5,6},{0,0}});     // anonymous array
        m2.setElement(1, 0, 7);                               // args: row, column, value
        m2.setElement(1, 1, 8);

        System.out.println("Matrix m1:");
        System.out.println(m1);                               // invoke m1.toString() implicitly

        System.out.println("Matrix m2:");
        System.out.println(m2);

        System.out.println("Result of m1 + m2 -> m1:");
        if(m1.add(m2))                                        // the sum is stored in m1
            System.out.println(m1);
        else
            System.out.println("Invalid matrix size.");


        System.out.println("Result of m1 x m2:");
        Matrix m3 = m1.multiply(m2);
        if(m3 != null)
            System.out.println(m3);
        else
            System.out.println("Invalid matrix size.");
    }
}
```

```
Output - JLab (run)
run:
Matrix m1:
[ 1][ 2]
[ 3][ 4]

Matrix m2:
[ 5][ 6]
[ 7][ 8]

Result of m1 + m2 -> m1:
[ 6][ 8]
[10][12]

Result of m1 x m2:
[86][100]
[134][156]

BUILD SUCCESSFUL (total time: 0 seconds)
```

6. Many scientific applications need to support arithmetic operations with numbers in a very large scale. In Java, the long data type is a 64-bit two's complement integer. The signed long has a minimum value of -2^{63} and a maximum

value of 2^{63}-1. In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 2^{64}-1.

Run the following program to see how overflow would occur.

```
public class TestBigDecimal {

    public static void main(String[] args) {

        long max = Long.MAX_VALUE;
        System.out.println("The maximum value represented by Long type is: " + max);
        System.out.println("Adding one to 'max' results in overflow: " + (max + 1));

    }

}
```

To address the need, Java provides a class *BigDecimal* that consists of an arbitrary precision integer unscaled value and a 32-bit integer scale which is the number of digits to the right of the decimal point. For example,

19/100 = 0.19 // unscaled value=19, scale=2

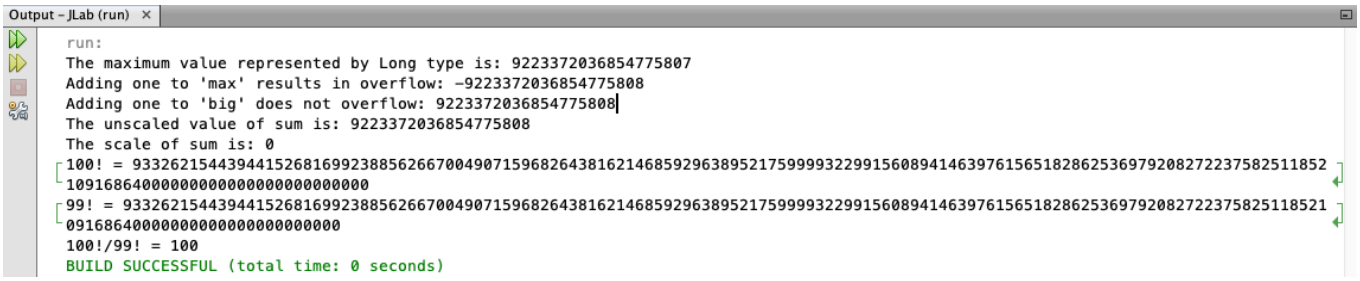
Run the following program to see how to use *BigDecimal* to handle arithmetic operations with big numbers.

```
public class TestBigDecimal {

    public static void main(String[] args) {

        long max = Long.MAX_VALUE;
        System.out.println("The maximum value represented by Long type is: " + max);
        System.out.println("Adding one to 'max' results in overflow: " + (max + 1));

        BigDecimal big = new BigDecimal(max);
        BigDecimal one = new BigDecimal(1);
        BigDecimal sum = big.add(one);
        System.out.println("Adding one to 'big' does not overflow: " + sum);
        System.out.println("The unscaled value of sum is: " + sum.unscaledValue());
        System.out.println("The scale of sum is: " + sum.scale());
    }

}
```

You are asked to implement the following method to compute the factorial of n:

```
public static BigDecimal factorial(long n);
```

To test your implementation, try to print out the value of 100! and 99! and the result of 100!/99!.



- END -