Object-Orientation & Inheritance

Table of Contents

1. Session 6 (lab): Object-Orientation & Inheritance (III)

1.1. Points and Figures (II)

The general objective of this lab is to recap inheritance and polymorphism in Java. You will be using `abstract` and `extends` for inheritance, creating generic and specific classes in a hierarchy. Specific classes will implement the abstract methods defined in the generic ones and will override the rest of their methods to implement polymorphism. The class hierarchy in this lab is composed by the superclass `Figure` and three of its children: `Circle`, `Triangle` and `Quadrilateral`. There is also a subclass of `Quadrilateral` called `Rectangle`.

Once you have implemented all these classes, you will have to create a bunch of objects and insert them into a container. Then you will perform operations on all of them taking advantage of polymorphism.

Abstract Classes and Inheritance.

The `Figure` class represents a generic figure, that will later be particularized into a more concrete class like `Circle`, `Triangle` or `Rectangle`. In Java, this kind of generic, abstract classes, are defined using the `abstract` reserved word.

In this first exercise, you must define the `Figure` abstract class. You can download an initial template for this class from here.

```
public abstract class Figure {

    // Name of the figure
    private String name;

    // Constructor of the figure
    public Figure(String name) {

    }

    // Calculates the area of a figure
    abstract public double area();

    // Indicates if the figure is regular or not
    abstract public boolean isRegular();

    // Gets the name of the figure
    protected String getName() {

    }

    // Sets the name of the figure
    protected void setName(String name) {

    }

}
```

The class `Figure`.

1. Program the constructor of the class `Figure` and its get and set methods.

The subclass `Circle`**.**

The class `Circle` inherits from `Figure`, take this into account when implementing it. Follow these steps:

1. Declare the class `Circle` inheriting from `Figure`, using `extends`.
2. Implement its constructor. Said constructor must accept three arguments: its name (`String`), its center (`Point`) and its radius (`int`). Use the class Point.java to represent points, and call the constructor of the `Figur` class through the `super` reference on the `Circle`'s constructor.
3. Program the method `area` which returns the area of the circle. Use the constant Math.PI in your calculations.
4. Program the method `isRegular()` which returns `true` if the figure is regular or `false` if it is not. Please note that circles are always regular figures.
5. Override the method `toString()` to return a textual representation of the name of the figure, its center and its radius (the format of this textual representation is up to you).
6. Program the get and set methods.

The subclass `Triangle`**.**

The class `Triangle` also inherits from `Figure`. Follow these steps:

1. Declare the `Triangle` class to inherit from `Figure`.
2. The constructor will receive four arguments: the name of the class and three points (`Point v1`, `Point v2` and `Point v3`), representing the three vertexes of the triangle. Remember you have to call the constructor of `Figure` using `super`.
3. Program its `area` method. The area of a triangle can be calculated form its vertexes using the following expression: `area = 0.5*|Ax(By-Cy)+Bx(Cy-Ay)+Cx(Ay-By)|`, where `x` is the abscissa and `y` the ordinate of the corresponding vertexes (A, B, C). In the expression, `|...|` represents the absolute value of what is inside.
4. Program the method `isRegular()` to tell if the triangle is regular. A triangle is regular if all its side lengths are equal.
5. Override the `toString` method to return a sensible textual representation of a triangle, including at least its name and its vertexes.
6. Program the get and set methods.

The subclass `Quadrilateral`**.**

The `Quadrilateral` is another subclass of `Figure`. But this time, we will make it also an abstract class, because it is difficult to calculate its area without knowing a little bit more about its details. Follow these steps:

1. Declare an abstract class `Quadrilateral` that inherits from `Figure`.
2. Implement its constructor, which receives five arguments: its name and four points.
3. Program the `isRegular` method to tell if the figure is regular. Please note that a quadrilateral is regular if it is also a rectangle. Use the method `checkRectangle` at the end of this section to help you program the `isRegular` method. How does the `checkRectangle` method works?
4. Override the `toString` method to return a sensible textual representation of a quadrilateral, including its name and its vertexes.
5. Program the get and set methods.

```
/**
 * Indicates if the quadrilateral is a rectangle or square by
 * comparing sizes and diagonals and calculating the scalar product
 *
 */
private boolean checkRectangle(Point v1, Point v2, Point v3, Point v4){
    Point auxVertex = v1.nearest(new Point[]{v2,v3,v4});
    if (auxVertex.equals(v2)){
        return v1.distance(v3) == v2.distance(v4)
            && v1.distance(v4) == v2.distance(v3)
            && scalarProduct(v1,auxVertex,v1.nearest(new Point[]{v3,v4}));
    } else if (auxVertex.equals(v3)){
        return v1.distance(v2) == v3.distance(v4)
            && v1.distance(v4) == v3.distance(v2)
            && scalarProduct(v1,auxVertex,v1.nearest(new Point[]{v2,v4}));
    } else if (auxVertex.equals(v4)){
        return v1.distance(v2) == v4.distance(v3)
            && v1.distance(v3) == v4.distance(v2)
            && scalarProduct(v1,auxVertex,v1.nearest(new Point[]{v2,v3}));
    } else {
        return false;
    }
}

private boolean scalarProduct(Point p1, Point p2, Point p3){
    return (p3.getY()-p1.getY())*(p2.getY()-p1.getY())
        + (p3.getX()-p1.getX())*(p2.getX()-p1.getX()) == 0;
}
```

The subclass `Rectangle`.

The subclass `Rectangle` inherits from `Quadrilateral`. Follow these steps:

1. Declare the class `Rectangle` that inherits from `Quadrilateral`.
2. Program its constructor, receiving 5 arguments, its name and its 4 vertexes. Remember to call the constructor of the `Quadrilateral` class through the `super` reference. The constructor must check that the four vertexes provided as its arguments are really the vertexes of a rectangle, not just of a generic quadrilateral. The constructor must throw an exception in case the vertexes are wrong (you can read about exceptions in this short introduction to exceptions).
3. Program its `area` method. In a rectangle, given a vertex, the distance to its two nearest vertexes, would be the base and height of the rectangle.
4. Override the `toString` method to return a sensible textual representation of a rectangle, including its name and its vertexes. Can you reuse some code from the `Quadrilateral.toString` method in here?

Polymorphism. Calculating the area of several figures.

The class `FiguresSet` will contain several figures, stored in an `ArrayList<Figure>` and will be able to tell the total area of the figures it holds. Here is a template for this class:

```
import java.util.ArrayList;

public class FiguresSet {

    // Array of figures
    private ArrayList<Figure> figures;

    // Constructor of FiguresSet
    public FiguresSet() {

    }

    // Calculates the total area of the figures
    public double totalArea() {

    }

    // figures to String
    public String toString() {

    }

    // Adds a new figure to the FigureSet
    public void addFigure(Figure f) {

    }

    // Main program
    public static void main(String args[]) throws Exception {

    }
}
```

The class constructor must create an object of the `ArrayList` class to store the figures. Then the user will be adding figures using the `addFigure` method. The method `totalArea` can then be used to get the total area of all the figures that have been added. The `toString` method returns a sensible textual representation of all the figures included in the `FigureSet`.

1. Implement the class constructor.
2. Write the `addFigure` method that allows to add a `Figure` to the container.
3. Write the `totalArea` method that allows to calculate the total area of the figures contained.
4. Write the `toString` method that returns a sensible textual representation of all the figures inside.

The `main` method.

Create the main program that has to carry out the following tasks:

1. Create an object of the `FiguresSet` class.
2. Create a circle.
3. Create a triangle.
4. Create a rectangle.
5. Add the circle, the triangle and the rectangle to the `FiguresSet` object.
6. Print the result of the calculation of the total area of the figures along with the information about the figures.

2. Homework

2.1. Using Interfaces to calculate Pi

Interfaces can be very useful for having several alternative implementations of the same functionality: we would like the main program to use one or the other without noticing the changes.

In this exercise, we will have several implementations of a class that calculates the number pi. We will be using the following interface:

```
import java.math.BigDecimal;

/**
 * Interface to be implemented by classes that compute the number pi.
 *
 */
public interface PiProvider {

    /**
     * Computes and returns the value of the number pi. Implementations
     * may decide the precision with which they compute the value.
     *
     * @return The number pi, with the precision each implementation
     *         decides.
     *
     */
    BigDecimal computePi();
}
```

As some implementations will be able to calculate pi with a desired precision, we will define also the following interface, that inherits from `PiProvider` and add some more methods to deal with precisions.

```
import java.math.BigDecimal;

/**
 * Interface to be implemented by classes that compute the number pi.
 * Classes that implement this interface can provide the value of pi
 * with the requested precision, understood as the number of exact
 * digits of the computed value.
 *
 */
public interface AdvancedPiProvider extends PiProvider {

    /**
     * Sets the desired precision.
     *
     * @param precision The desired precision (number of digits).
     *
     * @throws PrecisionException if the desired precision is
     *         negative, zero or bigger than the maximum precision
     *         this class can provide.
     *
     */
    void setPrecision(int precision) throws PrecisionException;

    /**
     * Returns the current value of precision.
     *
     * @return The current value of precision (number of digits).
     *
     */
    int getPrecision();

    /**
     * Returns the maximum precision with which this provider is able
     * to generate the value of pi.
     *
     * @return The maximum precision available from this provider, or
     *         Integer.MAX_VALUE if the provider can provide an
     *         arbitrarily big precision.
     *
     */
    int getMaximumPrecision();
}
```

As you can see, the method `setPrecision` will throw an exception if the desired precision is not a valid precision value (less than 1 or greater than the maximum precision allowed by each particular implementation). Here is the code of this exception:

```
public class PrecisionException extends Exception {
    public PrecisionException(int precision) {
        super("Unsupported precision: " + precision);
    }
}
```

Section 1

Program and test a class called `PiSimple` that implements the `PiProvider` interface and returns the value of pi as (simply) 3.14.

Section 2

Program and test a class called `PiFromMath` that implements the `PiProvider` interface and returns the value of pi defined by `Math.PI` as a `BigDecimal`.

Section 3

Program and test a class called `PiFromBBP` that implements the `AdvancedPiProvider` interface, using the code from the `PiCalc` class you wrote in a previous lab. You will only need to make a few changes to adapt it.

```
import java.math.BigDecimal;
import java.math.MathContext;

public class PiCalc {

    private int numDigits;
    private MathContext mc;

    public PiCalc(int numDigits) {
        this.numDigits = numDigits;
        mc = new MathContext(numDigits);
    }

    public BigDecimal compute() {
        BigDecimal pi = new BigDecimal(0);
        BigDecimal limit = new BigDecimal(1).movePointLeft(numDigits);
        boolean stop = false;
        for (int k = 0; !stop; k++) {
            BigDecimal piK = piFunction(k);
            pi = pi.add(piK);
            if (piK.compareTo(limit) < 0) {
                stop = true;
            }
        }
        return pi.round(mc);
    }

    private BigDecimal piFunction(int k) {
        int k8 = 8 * k;
        BigDecimal val1 = new BigDecimal(4);
        val1 = val1.divide(new BigDecimal(k8 + 1), mc);
        BigDecimal val2 = new BigDecimal(-2);
        val2 = val2.divide(new BigDecimal(k8 + 4), mc);
        BigDecimal val3 = new BigDecimal(-1);
        val3 = val3.divide(new BigDecimal(k8 + 5), mc);
        BigDecimal val4 = new BigDecimal(-1);
        val4 = val4.divide(new BigDecimal(k8 + 6), mc);
        BigDecimal val = val1;
        val = val.add(val2);
        val = val.add(val3);
        val = val.add(val4);
        BigDecimal multiplier = new BigDecimal(16);
        multiplier = multiplier.pow(k);
        BigDecimal one = new BigDecimal(1);
        multiplier = one.divide(multiplier, mc);
        val = val.multiply(multiplier);
        return val;
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("One command-line argument expected: number of "
                               + "digits.");
        } else {
            PiCalc piCalc = new PiCalc(Integer.parseInt(args[0]));
            System.out.println(piCalc.compute());
        }
    }
}
```

Please note that the new constructor will not receive any arguments, as the default desired precision will be 30 decimal places.

Section 4

Program and test a class called `PiStored` that stores a precalculated value of pi as a `String`. When asked for a new value of pi, it will return the corresponding stored value (using the desired precision). The maximum precision will be limited by the size of the stored string.

Its constructor must not receive any parameters. The default precision will be 30 decimal places.

You can use the following string as the value of pi:

```
private static final String PI = "3.14159265358979323846264338327950"
                                + "28841971693993751058209749445923O7"
                                + "8164062862089986280348253421170679";
```

To copy a piece of the string, you can use the method `substring` from the class `String`.

Section 5

Program and test a class called `Circle` that:

- Stores its radius as a `BigDecimal`.
- Has a constructor that receives the radius.
- Has a method called `area` that receives an object implementing the `PiProvider` interface and returns the area of the circle calculated using the value of pi supplied by that object.

You must carefully make use of polymorphism so this class can use any implementation of the `PiProvider` interface. This means also to foresee new implementations, different from the ones you made yourself. Test your code using different instances of each of your interface implementations.