jv-oop-advanced

Solve the task using OOP principles

Task:

There are some figures of the following types: square, rectangle, right triangle, circle, isosceles trapezoid. You need to create corresponding classes for them(`Square`, `Rectangle`, `RightTriangle`, `Circle`, `IsoscelesTrapezoid`)

All figures have

- **state** - all figures have `color`, but each figure type can also have one or several unique properties (`radius` for circle, `firstLeg` and `secondLeg` for right triangle, and so on).
- **behavior** - we can obtain the area of any figure and are able to draw it. To 'draw' means to print out all information about a figure using `System.out.println()` (you shouldn't override the toString() method for this).

Think where you should declare these fields and methods: top-level class/interface / bottom-level classes.

In the `main()` method we need to create an array of figures (the size of the array can be 3 or 6, it doesn't matter). **The first half** of figures in this array should be generated with random parameters.

For this purpose create two more classes:

- `ColorSupplier` with `public String getRandomColor()` method - for generating random color,
- and `FigureSupplier` with the `public Figure getRandomFigure()` method - for generating figures with random properties.

The other half of the figures should have the same, default parameters.

For this purpose create a new method in the `FigureSupplier` class:

- `public Figure getDefaultFigure()` - this method should always return a white circle with a radius of 10.

After generating the array, we need to display the entire list of objects that we have, for example:

```
Figure: square, area: 25.0 sq. units, side: 5 units, color: blue

Figure: triangle, area: 12.5 sq. units, firstLeg: 7 units, secondLeg: 5 units, color: yellow
```

Don't begin class or method implementation with an empty line.

Remove all redundant empty lines, be careful :)

Don't use abstract classes to set behavior for classes

Abstract classes and interfaces have different use cases. Try to figure out when to use both in this task by yourself. If you're blocked this may give you a hint.

Don't use verbs for class/interface names

- Bad example:

```
public interface CalculateArea {
}
```

- Improved example:

```
public interface AreaCalculator {
}
```

Don't put all behavior into a single interface if the methods are conceptually different from each other.

All our classes and interfaces should have a single purpose - the `draw()` and `getArea()` methods are not conceptually close to each other.

You can pass random values to the constructor of a figure instead of generating them inside figure classes.

Let's generate random values in `FigureSupplier`.

Think about which variables should be local in the method and which should be class-level

- Bad example:

```
public class AccauntService {
    public int calculateTax(int income) {
        TaxService taxService = new TaxService();
        int tax = taxService.getTax();
        return income * tax / 100;
    }
}
```

- Improved example:

```
public class AccauntService {
    private TaxService taxService = new TaxService();

    public int calculateTax(int income) {
        int tax = taxService.getTax();
        return income * tax / 100;
    }
}
```

All magic numbers in your code should be constants.

Please see this article to learn about constant fields and their naming requirements.

- Bad example:

```
public class FigureSupplier {
    private Random random = new Random();

    public Figure getRandomFigure() {
        int `figureNumber` = random.nextInt(5);
        // generate a specific figure based on the `figureNumber` value
    }
}
```

- Improved example:

```
public class FigureSupplier {
    public static final int FIGURE_COUNT = 5;
    private Random random = new Random();

    public Figure getRandomFigure() {
        int figureNumber = random.nextInt(FIGURE_COUNT);
        // generate a specific figure based on the `figureNumber` value
    }
}
```

Creating a figure, don't pass expressions in the constructor.

Create separate variables and pass them on for better code readability.

- Bad example:

```
Square square = new Square(random.nextInt(10) + 1);
```

Don't use static methods in your solution

Static methods are in general a bad practice. Let's better create an instance of a class which method you want to call.

Don't extend your `Main/Application` class from `FigureSupplier` or `ColorSupplier`.

To be able to call the non-static method, we just need to create an instance of the class:

```
FigureSupplier figureSupplier = new FigureSupplier();
Figure randomFigure = figureSupplier.getRandomFigure();
```

You should create several random Figures, so you will use a loop. Please don't create a `new FigureSupplier()` inside the loop.

Let's do it only once - before the loop starts.

Don't return `null` from a method.

Returning `null` from a method is a bad practice. If you use a `switch case construction` in your solution, you may just put the last possible option in the `default` case.

Use only eng in messages/code:

Try not to use ukr/ru messages in `toString()` or `System.out.println()` statements. We want to make our code universal and consistent.

Use name() for getting String representation of enum constants

Don't use `toString()` or `String.valueOf()`(it will call `toString()` under the hood) for getting the `String` representation of enum constants. `toString()` is common for all enum constants. If you override this method like below:

```
@Override

public String() toString() {

    return "default";

}
```

then for every constant `toString()` will be returning `default`, that's not ok. So it's better to use the standard method of enum `name()` that will be returning always `String` representation of the concrete enum constant.

Write informative messages when you commit code or open a PR.

Bad examples of commit/PR messages: `done/fixed/commit/solution/added homework/my solution` and other one-word, abstract or random messages.