Lab-7: GoF Design Patterns - Decorator

1 Purpose

The Decorator Pattern serves to decorate, extend, or wrap an existing object with additional functionality. The challenge that you may sometimes face is that you have concrete objects to which you want to add functionality, but:

- You may not want to add it to all objects in the inheritance hierarchy, or
- It may be extrinsic to the business functionality of the object in question, such as logging, performance monitoring, or access controls.

For example, if you have a base `Widget` class and you've derived four classes from it: `WhiteWidget`, `YellowWidget`, `BlueWidget`, and `BlackWidget`, and you want to add new functionality for dark and light widgets, what do you do?

You don't want to add the functionality to the base Widget class because not all Widgets are dark and not all widgets are light.

You could create `DarkBlueWidget` and `DarkBlackWidget` classes and copy the “dark” code between them, but that means that you'll have duplicate code. Furthermore, if you add a new dark Widget, you'll have to copy the code again.

One perfectly good solution to this example is to create `DarkWidget` and `LightWidget` classes that extends `Widget` and then extend the other widgets from them, but what if we also want to add functionality that only applies to primary color Widgets?

This functionality only applies to the `YellowWidget` and `BlueWidget`, but the `YellowWidget` is a `LightWidget` and the `BlueWidget` is a `DarkWidget`, so now our inheritance hierarchy is getting more complex.

Although this is a contrived example, it illustrates a case when we want to add external functionality to objects that may have no business being in the inheritance hierarchy.

Lab-7: GoF Design Patterns - Decorator

The better solution is to allow a widget to be a widget and create a set of **decorators** that know how to make a widget dark or light or to handle primary colors. A contrived color example is one thing, but the best way to fully understand the Decorator Pattern is to see it in a practical example:

2 What to do?

For this exercise, We will build a `interface MathOperation` that exposes a single method: `public Double eval(Double... operands)` that accepts a variable number of operands.

From the `MathOperation` we derive two implementations: `AddOperation` and `MultiplyOperation` (and possibly other operations).

Then, we will create `abstract class MathOperationDecorator` that serves as the base class for all decorators, and two decorator implementation classes: `class LoggingDecorator` and `class PerformanceDecorator`.

The `class LoggingDecorator` prints the name of the `MathOperation` class, its arguments, and its results to the standard output device and the `PerformanceDecorator` records the time the operation takes to execute, and prints that to the standard output device.

Finally we provide a test application that shows the two operations running without decoration, decorated by the `LoggingDecorator`, decorated by the `PerformanceDecorator`, and decorated by both.

* **Note** all of the classes described are provided for you in `package patterns.structural.decorator`. Some, require code completion on your behalf, as instructed bellow and is demarcated by `// TODO:` tags in the code.

Other classes are fully provided, including the test class.

Lab-7: GoF Design Patterns - Decorator

3 How to do it?

#	What to do?	How to do it.
1	***Review the provided code***	
A	Review the interface `MathOperation`. this interface is provided for you and defines the basic behavior of all `MathOperation` implementations	* The single method `eval` accepts a variable length argument of doubles (ellipsis): `public Double eval(Double... operands);` and returns a `Double` result suitable for all math operations including division.
B	Review the provided `AddOperation` class. This is our provided implementation of the `MathOperation` interface. You can use this example to implement the next concrete implementation, the class `MultiplyOperation`	* The mandatory `eval` method simply iterates over the double arguments and sums them up (possibly using a stream operation).
C	Review the provided `abstract class MathOperationDecorator`. This is a base class for all decorators in this lab.	* The only thing it performs is a consisting a placeholder for the current composed operation (e.g. being a wrapper for a 'wrapee' operation) and to define the overloaded constructor which accepts a `MathOperation` argument. You will need to complete the `LoggingDecorator` class and the class `PerformanceDecorator` which, both, extend `MathOperationDecorator`, to add the required enhanced decorated functionality.
D	The `class DecoratorDemo` is provided here to save you the time of writing a test. It compiles and works but...	* The demo compiles and works but is meaningless before you complete this lab and implement the required functionality.
2	***Implement the `MathOperation` class***	
A	Use the `AddOperation` class as an example to implement a multiplication operation class.	This should be too easy. All you need to do is modify the operator. The implemented class should look like this:

Lab-7: GoF Design Patterns - Decorator

#	What to do?	How to do it.

	`public class MultiplyOperation implements MathOperation {`
	
	`    @Override`
	`    public Double eval(Double... operands) {`
	`        Double result = 1D;`
	`        for (Double operand : operands)`
	`            result *= operand;`
	`        return result;`
	`    }`
	`}`	

B	* In the same way, you can implement the `SubtractionOperation` and `DivisionOperation` classes as an extra exercise.	
3	***Implement the `LoggingDecorator` for Logging Enhancement***	
A	The `LoggingDecorator` class prints the name of the `MathOperation` class, its arguments, and its results to the standard output device	This class **IS-A** `MathOperationDecorator`. As such it must implement the `public Double eval(Double... operands)` method dictated by the super `MathOperation` interface. Naturally, at some point in the implementation we will witness some sort of delegation of the actual work to the wrapped `MathOperation`. (well, composition is a major idiom of the decorator pattern). But, before and after the actual delegation we can output some information the standard out. We can also 'pretty print' the output, so an implementation of the `eval` method, may look like this:

```java
    @Override
    public Double eval(Double... operands) {
      Double result = 0D;
      // Before the evaluation: Log the operation name and arguments
      StringBuilder sb = new StringBuilder(operands.length);
      sb.append("[");
      for (Double operand : operands)
          sb.append(operand).append(",");
      sb.replace(sb.lastIndexOf(","), sb.lastIndexOf(",") + 1, "]");
      System.out.printf("LoggingDecorator.eval() -> evaluating %s with
            operands %s%n", operation.getClass().getSimpleName(), sb.toString());
      result = operation.eval(operands);
      // After the evaluation: Log the result
      System.out.printf("LoggingDecorator.eval() -> %s %s = %s%n",
            sb.toString(), operation.getClass().getSimpleName(), result);
      return result;
    }
```

Lab-7: GoF Design Patterns - Decorator

#	What to do?	How to do it.
4	***Implement the `PerformanceDecorator` to achieve profiling functionality***	
A	You've already implemented one decorator. The next, is suppose to take the time before the delegation of the actual work to the wrapee operation, and after the delegation and then pretty print the difference in milli-seconds.	This is basic profiling 101. Real easy. Once again we perform actions both before and after the actual delegation. Take a look at one possible implementation bellow:

```java
    @Override
    public Double eval(Double... operands) {
        long startTime = System.nanoTime();
        Double result = operation.eval(operands); // Execute the operation
        long endTime = System.nanoTime(); // Measure the execution time
        System.out.printf("PerformanceDecorator,eval() -> executed %s in %.2fms
           %n",operation.getClass().getSimpleName(), (endTime startTime) / 1000D);
        return result;
    }
```

#	What to do?	How to do it.
5	***Run the `DecoratorDemo` test class***	

A	Now that the complete exercise is complete, you can run the example class and test the functionality of your decorators.	* You are **not** suppose to witness any
		
		`“I'm suppose to log the result...”`
		or
		`“I was supposed to take the time after the operation execution...”`
		
		messages. These only exist in the provided code to remind you that you are suppose to replace them with your own implementation.
		
		The outcome should look something like this:

Lab-7: GoF Design Patterns - Decorator

#	What to do? How to do it.
	```
	No Decorators
	=============
	100 + 100 = 200.00
	100 * 100 = 10000.00

	Logging Decorator
	=================
	Math Operation: patterns.structural.decorator.AddOperation with args: 100.00,
	100.00 returned result: 200.00
	100 + 100 = 200.00
	Math Operation: patterns.structural.decorator.MultiplyOperation with args: 100.00,
	100.00 returned result: 10000.00
	100 * 100 = 10000.00

	Performance Decorator
	=====================
	Execution for operation patterns.structural.decorator.AddOperation took 9.51ms to
	execute
	100 + 100 = 200.00
	Execution for operation patterns.structural.decorator.MultiplyOperation took
	2.26ms to execute
	100 * 100 = 10000.00

	Performance and Logging Decorator
	=================================
	Math Operation: patterns.structural.decorator.AddOperation with args: 100.00,
	100.00 returned result: 200.00
	Execution for operation patterns.structural.decorator.LoggingDecorator took
	167.99ms to execute
	100 + 100 = 200.00
	Math Operation: patterns.structural.decorator.MultiplyOperation with args: 100.00,
	100.00 returned result: 10000.00
	Execution for operation patterns.structural.decorator.LoggingDecorator took
	147.16ms to execute
	100 * 100 = 10000.00
	```

4 Summary

Observations that you can make from this are:

- The `MathOperationDecorator` can decorate any `MathOperation` that you create. For example you might create a `SubtractOperation` or `DivideOperation` and you do not need to worry about integrating logging or performance functionality into it, you can simply decorate it with one of the decorators

- There is a strong separation of concerns: math operations should only be concerned with performing math operations, not with logging or performance measurements. The Decorator Pattern promotes the modern "**Separation of Concerns**" design principle.

- Decorators can wrap decorators: you can chain together as many decorates as you want. This is possible because each decorator contains a reference to the undecorated object to which it delegates the actual implementation.

- The Decorator Design Pattern promotes the notion of separation of concerns by externalizing object functionality into wrappers called decorators. A decorator wraps an object and provides some additional, and usually cursory, functionality to that object. It solves the problem of selectively adding functionality to an object without disrupting its inheritance hierarchy.