Chapter 4

What is the Decorator Pattern?

The Decorator Pattern is a structural design pattern that:

Lets you add new behavior or responsibilities to objects dynamically (at runtime).

Does this without modifying the original class or creating endless subclasses.

👉 In simple terms: It's like wrapping a gift box inside multiple layers of wrapping paper — each layer adds something extra.

⚡ Why use it?

Avoids class explosion (too many subclasses for every combination of features).

Provides flexibility to extend object functionality at runtime.

Follows the Open/Closed Principle (open for extension, closed for modification).

❌ Without Decorator (Bad Example)

```
class Coffee {
  public String getDescription() {
    return "Simple Coffee";
  }
  public double getCost() {
    return 5.0;
  }
}

class MilkCoffee extends Coffee {
  @Override
  public String getDescription() {
    return "Simple Coffee + Milk";
  }
  @Override
  public double getCost() {
    return 7.0;
```

```
    }
}

class SugarMilkCoffee extends Coffee {
    @Override
    public String getDescription() {
        return "Simple Coffee + Milk + Sugar";
    }
    @Override
    public double getCost() {
        return 8.0;
    }
}
```

🔴 Problem: Every new combination (milk, sugar, caramel, etc.) needs a new subclass → leads to class explosion.

✅ With Decorator Pattern (Good Example)

```
// Step 1: Component
interface Coffee {
    String getDescription();
    double getCost();
}

// Step 2: Concrete Component
class SimpleCoffee implements Coffee {
    public String getDescription() {
        return "Simple Coffee";
    }
    public double getCost() {
        return 5.0;
    }
}

// Step 3: Base Decorator
abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee; // wrap the coffee object
    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }
```

```
    public String getDescription() {
        return coffee.getDescription();
    }
    public double getCost() {
        return coffee.getCost();
    }
}

// Step 4: Concrete Decorators
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }
    public String getDescription() {
        return super.getDescription() + " + Milk";
    }
    public double getCost() {
        return super.getCost() + 2.0;
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }
    public String getDescription() {
        return super.getDescription() + " + Sugar";
    }
    public double getCost() {
        return super.getCost() + 1.0;
    }
}
```

✅ Client Code (runtime flexibility)

```
public class Main {
    public static void main(String[] args) {

        Coffee coffee=new SugarDecorator(new MilkDecorator(new Simple
            Coffee())); // add MILK and SUGAR

        System.out.println(coffee.getDescription() + " = $" + coffee.getCost());
```

//NOW IF WE WANT OUR COFFEE TOP HAVE BIOTH MILK AND SUGAR, WE WILL WRITE FOLLOWING

```
  }
}
```

✅ Output

```
Simple Coffee = $5.0
Simple Coffee + Milk + Sugar = $8.0
```

First create coffee interface with method cost

Then create simplecoffee class which implements the above interface.

Now create a abstract class(BaseDecorator) which implements the coffee interface . this class will also have reference of coffee interface.
It will have a constructor where this reference will be initialized.and it will implement the methods present in coffee interface.

Now create a concrete class (MilkDecorator) which extends this abstract class (BaseDecorator). Create the constructor and methods to call super class constructor and methods.

Now create a main class here create a object of simplecoffee in following way with decoration:

```
Coffee coffee=new SugarDecorator(new MilkDecorator(new Simple     Coffee()));
// add MILK and SUGAR
//and now call the cost method to detect the cost
coffee.getCost();
```