Chapter 2

What is Strategy Pattern?

Think of it like choosing different ways to travel to work:

- You can take a **car**
- You can take the **bus**
- You can take the **train**
- You can **walk**

All of these are different **strategies** to reach the same goal (getting to work), but you can choose which one to use based on the situation.

Real Life Example

Imagine you have a **Calculator** that needs to do math operations. Instead of writing all the math inside one big calculator class, you create separate "strategy" classes for each operation.

java

```java
// This is like saying "every math operation must have a calculate method"
interface MathStrategy {
  int calculate(int a, int b);
}

// Different ways to do math (different strategies)
class AddStrategy implements MathStrategy {
  public int calculate(int a, int b) {
    return a + b;
  }
}

class SubtractStrategy implements MathStrategy {
  public int calculate(int a, int b) {
    return a - b;
  }
}

class MultiplyStrategy implements MathStrategy {
  public int calculate(int a, int b) {
    return a * b;
  }
}
```

```
// The calculator that uses different strategies
class Calculator {
  private MathStrategy strategy;

  // Change the strategy (like changing your travel method)
  public void setStrategy(MathStrategy strategy) {
    this.strategy = strategy;
  }

  // Do the calculation using current strategy
  public int doMath(int a, int b) {
    return strategy.calculate(a, b);
  }
}

// How to use it
public class Example {
  public static void main(String[] args) {
    Calculator calc = new Calculator();

    // Use addition strategy
    calc.setStrategy(new AddStrategy());
    System.out.println("5 + 3 = " + calc.doMath(5, 3)); // Output: 8

    // Switch to multiplication strategy
    calc.setStrategy(new MultiplyStrategy());
    System.out.println("5 * 3 = " + calc.doMath(5, 3)); // Output: 15
  }
}
```

Why Use This Pattern?

1. Easy to Add New Ways

- Want to add division? Just create a new DivideStrategy class
- No need to change the existing calculator code

2. Easy to Switch

- You can change from addition to multiplication anytime
- Like switching from car to bus when there's traffic

3. Clean Code

- Each math operation has its own small, simple class
- No big messy if-else statements

Simple Analogy

Think of it like a **TV remote**:

- The TV (Context) stays the same
- But you can change channels (Strategies)
- Each channel shows different content (Different behavior)
- You decide which channel to watch (Which strategy to use)

When Should You Use This?

Use Strategy Pattern when you have:

- Multiple ways to do the same thing
- You want to switch between these ways easily
- You don't want one big messy class with lots of if-else statements

Examples:

- Payment methods in online shopping (Credit card, PayPal, Bank transfer)
- Different discounts (Student discount, Senior discount, Holiday discount)
- Game character behaviors (Aggressive, Defensive, Sneaky)

For my personal reference:

1)Adhi ek interface bnanva with a method say start();

2)mg classes banva je tya interface extend kartat ani toh ek method je interface mdhe hota te aplya swatachya prakare modify krtat.

3) ek class bnva tya class mdhe aplya interface ch instance bnva . ani constructor mdhe tya interface la instantiate kara. Like vehicle(Vehicle v){this.v=v;} . ani ek method bnva jya mdhe aplya interface ch method apn aplya object chy madatine bolvu shakto. Example
Public void run(){this.v.start()}

4)Mg main class bnva ani ikde aplya 3rd step mdhe bnvlela class ch object bnva say obj1 ani tyachya constructor mdhe ekhadya class from step 2 ch object pass kara.
Mg ata obj1.run() ; ha method bolva for execution