DESIGN PATTERN

What is a Design Pattern?

A **Design Pattern** is a **proven solution to a common software design problem**. It provides a structured way to write code that is **maintainable, reusable, and scalable**.

Think of it as a **blueprint** or a **best practice** that developers follow to solve recurring design issues efficiently.

Why Do We Need Design Patterns?

1. **Improves Code Reusability** – Common solutions can be applied across different projects.
2. **Enhances Maintainability** – Code becomes **structured and easy to understand**.
3. **Reduces Code Duplication** – Encourages reusing existing patterns instead of reinventing solutions.
4. **Supports Scalability** – Helps in designing software that can grow without major changes.
5. **Follows SOLID Principles** – Many patterns help implement SOLID principles effectively.

Types of Design Patterns (with Examples)

1. **Creational Patterns** – Focus on **object creation**.
    - **Singleton** – Ensures only one instance of a class (e.g., Logger, Database Connection).
    - **Factory Method** – Creates objects without specifying the exact class.
    - **Builder** – Helps create complex objects step by step.
2. **Structural Patterns** – Focus on **class and object composition**.
    - **Adapter** – Allows incompatible interfaces to work together.
    - **Decorator** – Adds new behavior dynamically.
3. **Behavioral Patterns** – Focus on **communication between objects**.
    - **Strategy** – Allows selecting an algorithm at runtime (e.g., different sorting techniques).
    - **Observer** – Used for event-driven programming (e.g., Notification System).

Key Takeaways

✔ **Design Patterns make software development easier and more efficient.**
✔ **They provide reusable solutions to common problems.**
✔ **Used in web development, automation testing, mobile apps, and more.**

SINGLETON CLASS PATTERN

What is the Singleton Pattern in Java?

The **Singleton Pattern** is a **design pattern** that ensures a class has **only one instance** and provides a **global access point** to that instance.

💡 **Example:** Imagine a **Printer Spooler** in a computer system. There should be **only one spooler instance** managing all print jobs. This is where the Singleton pattern is useful.

Why is Singleton Required?

✓ **Ensures a single instance** – Prevents multiple object creation, saving memory.
✓ **Provides global access** – The instance is accessible from anywhere in the application.
✓ **Useful for resource management** – Helps manage shared resources like database connections, logging, thread pools, etc.

Use Cases of Singleton Pattern

📌 **Database Connection Pool** – Ensures only one connection manager exists.
📌 **Logger Class** – Maintains a single logging instance across the application.
📌 **Configuration Manager** – Stores application settings globally.
📌 **Thread Pools** – Ensures a single instance manages threads efficiently.
📌 **Cache Management** – Prevents redundant cache object creation.

How to Implement Singleton in Java?

3 Thread-Safe Singleton (Using Synchronized)

```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

✅ **Thread-safe**, but **synchronization makes it slow**.

---

4 Double-Checked Locking (Best Approach)

```
class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}
```

```java
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

✅ **Efficient and thread-safe**. **Recommended for production use**.

Why volatile and synchronized are used in Double-Checked Locking Singleton?

◆ **Problem Without volatile and synchronized**

In a **multi-threaded environment**, if two threads try to create an instance at the same time, there is a risk of:
✅ **Multiple instances getting created** (breaking Singleton).
✅ **Half-initialized object being accessed** (due to reordering issues).

💡 **Solution**:

- **volatile** ensures proper visibility of the instance across threads.
- **synchronized** ensures only one thread enters the critical section at a time.

How to Answer in an Interview?

👉 **Short Answer:**
"volatile ensures **visibility and prevents instruction reordering**, while synchronized ensures **only one thread creates the instance** at a time. This makes the Singleton pattern **efficient and thread-safe** without unnecessary synchronization overhead."

How to Prevent Reflection Attack in the Singleton Pattern?

A reflection attack occurs when Java Reflection is used to break Singleton by accessing its private constructor and creating multiple instances. To prevent this, we can throw an Exception in the Constructor.

Throw an Exception in the Constructor

"We can modify the constructor to check if an instance already exists and throw an exception if an attempt is made to create a second instance."

Example:

```
private Singleton() {

    if (instance != null) {

        throw new RuntimeException("Reflection attack detected!");

    }

}
```

How to prevent if singleton implements serializable interface?

Serialization can break a Singleton by deserializing an object into a new instance. If a Singleton class implements Serializable, Java allows deserialization to create a new instance, bypassing the Singleton pattern. To prevent this, we override the readResolve() method.

Example:

```
// Prevents creating a new instance on deserialization

    protected Object readResolve() {

        return INSTANCE;

    }
```

To prevent a serialization attack in Singleton, we implement readResolve(), which ensures that during deserialization, the existing instance is returned instead of creating a new one. This effectively preserves Singleton behavior

How to prevent Cloning Attack in Singleton Pattern?

Cloning can break a Singleton if the class implements Cloneable. The clone() method in Object creates a new instance, violating the Singleton pattern. To prevent this, we override clone() and throw an exception.

Example:

```
 @Override

    protected Object clone() throws CloneNotSupportedException {

        throw new CloneNotSupportedException("Cloning is not allowed for Singleton!");

    }
```

To prevent cloning from breaking the Singleton pattern, we override the clone() method and throw a CloneNotSupportedException. This ensures no new instances are created via cloning

FACTORY PATTERN

1 What is the Factory Pattern?

💡 "Factory Pattern is a creational design pattern that provides an interface for creating objects but allows subclasses to decide which class to instantiate. It helps in achieving loose coupling by delegating object creation to a separate method."

2 How Can We Achieve the Factory Pattern?

💡 "We achieve the Factory Pattern by creating a factory class with a static method that returns different types of objects based on input parameters."

✅ **Example: Vehicle Factory**

// Step 1: Create an interface

```
interface Vehicle {
   void drive();
}
```

// Step 2: Implement different classes

```
class Car implements Vehicle {
   public void drive() { System.out.println("Car is driving"); }
}
class Bike implements Vehicle {
   public void drive() { System.out.println("Bike is driving"); }
}
```

// Step 3: Create Factory Class

```
class VehicleFactory {
   public static Vehicle getVehicle(String type) {
      if (type.equalsIgnoreCase("car")) {
         return new Car();
      } else if (type.equalsIgnoreCase("bike")) {
```

```
            return new Bike();
        }
        throw new IllegalArgumentException("Unknown vehicle type");
    }
}
```

// Step 4: Usage

```
public class FactoryPatternExample {
    public static void main(String[] args) {
        Vehicle vehicle1 = VehicleFactory.getVehicle("car");
        Vehicle vehicle2 = VehicleFactory.getVehicle("bike");

        vehicle1.drive(); // Car is driving
        vehicle2.drive(); // Bike is driving
    }
}
```

3 What Are the Uses of the Factory Pattern?

💡 "The Factory Pattern is useful in scenarios where object creation logic should be centralized and flexible. Some common use cases include:"

✅ **1. Web Automation (Selenium Browser Factory)**

☞ Creating WebDriver instances dynamically based on the browser type.

```
class WebDriverFactory {
    public static WebDriver getDriver(String browser) {
        if (browser.equalsIgnoreCase("chrome")) {
            return new ChromeDriver();
        } else if (browser.equalsIgnoreCase("firefox")) {
            return new FirefoxDriver();
        }
        throw new IllegalArgumentException("Invalid browser type");
    }
}
```

BUILDER PATTERN

What is the Builder Pattern?

The **Builder Pattern** is a **creational design pattern** used to create complex objects step by step. Instead of using a long constructor with many parameters, we use a builder class to make object creation **more readable and manageable**.

Why Do We Need the Builder Pattern?

1. **Avoids Long Constructors** – Helps when a class has many optional parameters.
2. **Improves Readability** – Code is cleaner and easier to understand.
3. **Encapsulates Object Creation Logic** – Keeps object setup separate from business logic.
4. **Supports Immutable Objects** – Ensures object properties cannot be changed after creation.

When to Use the Builder Pattern?

- When a class has **many optional parameters**.
- When you need **better readability** for object creation.
- When you want **immutable objects**.

Example:

```
class User {

   private String name;

   private int age;

   private String email;



   private User(Builder builder) {

      this.name = builder.name;

      this.age = builder.age;

      this.email = builder.email;

   }
```

```java
    public static class Builder {
        private String name;
        private int age;
        private String email;

        public Builder setName(String name) { this.name = name; return this; }
        public Builder setAge(int age) { this.age = age; return this; }
        public Builder setEmail(String email) { this.email = email; return this; }
        public User build() { return new User(this); }
    }
}

// Usage
User user = new User.Builder()
            .setName("John")
            .setAge(30)
            .setEmail("john@example.com")
            .build();
```

STRATEGY PATTERN

What is the Strategy Pattern?

The **Strategy Pattern** is a **behavioral design pattern** that allows you to define a family of algorithms, encapsulate each one in a separate class, and switch between them at runtime **without modifying the client code**.

Why Do We Need the Strategy Pattern?

1. **Avoids If-Else or Switch Statements** – Instead of writing multiple if-else conditions, we separate different behaviors into their own classes.
2. **Supports Runtime Flexibility** – You can change the behavior of a class **dynamically** at runtime.
3. **Follows Open-Closed Principle** – New strategies can be added **without modifying existing code**.

Example in Java

Let's say we have different payment methods (Credit Card, PayPal).

Step 1: Create a Strategy Interface

- This is the common interface that all payment strategies will implement.
- It defines a pay(int amount) method.

```
interface PaymentStrategy {

  void pay(int amount);

}
```

✅ **Why?**

- Ensures all payment methods follow a common structure.

Step 2: Implement Different Payment Strategies

- We create multiple classes that implement the PaymentStrategy interface.

```
class CreditCardPayment implements PaymentStrategy {

  public void pay(int amount) {

    System.out.println("Paid " + amount + " using Credit Card.");

  }

}



class PayPalPayment implements PaymentStrategy {

  public void pay(int amount) {

    System.out.println("Paid " + amount + " using PayPal.");

  }

}
```

✅ **Why?**

- Each payment method has its own implementation.
- You can add new payment methods **without modifying existing code**.

Step 3: Create a Context Class

- This class holds a reference to a **strategy (payment method)**.
- The client can change the strategy **at runtime**.

```
class PaymentContext {

    private PaymentStrategy strategy;



    public void setPaymentStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }



    public void payAmount(int amount) {

        strategy.pay(amount);

    }

}
```

✅ **Why?**

- Allows us to **dynamically switch strategies**.
- **Decouples payment logic** from the main business logic.

Step 4: Use the Strategy Pattern in the Main Method

- We create a PaymentContext object.
- We set different strategies (CreditCardPayment, PayPalPayment).
- We change the strategy **at runtime**.

```
public class Main {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();
```

```
        // Pay using Credit Card

        context.setPaymentStrategy(new CreditCardPayment());

        context.payAmount(100);



        // Switch to PayPal at runtime

        context.setPaymentStrategy(new PayPalPayment());

        context.payAmount(200);

    }

}
```

✅ **Why?**

- The payment method **can be changed at runtime** without modifying existing code.

◆ **Real-World Scenario**

Think of an **e-commerce website** where a user selects a payment method at checkout.
The website should not care **how** the payment is processed; it should simply execute the selected payment method.

Without the Strategy Pattern:

- You would have a **big if-else block** for different payment methods.
- Any new payment method would require **modifying** the existing code.

With the Strategy Pattern:

- You can **add new payment methods without modifying existing code**.
- The payment method **can be changed dynamically**.

🎯 **Key Takeaways**

✅ **Removes if-else conditions**
✅ **Supports dynamic behavior switching**
✅ **Follows Open-Closed Principle (OCP)** – New strategies can be added without modifying existing code.
✅ **Improves code readability and maintainability**

DECORATOR PATTERN

What is the Decorator Pattern?

The **Decorator Pattern** is a **structural design pattern** that allows adding new behaviors to objects **dynamically at runtime without modifying their original code**.

Instead of modifying an existing class, we wrap it inside another class (decorator) that adds additional functionality.

Why Do We Need the Decorator Pattern?

1. **Extends functionality without modifying existing code** (Follows Open-Closed Principle).

   Open-Closed Principle (OCP) - Simple Explanation

   The **Open-Closed Principle (OCP)** states that:

   "Software entities (classes, modules, functions) should be open for extension but closed for modification."

   This means:
   ✅ **You should be able to add new functionality without changing existing code.**
   ❌ **You should not modify existing working code when requirements change.**

2. **Avoids subclass explosion** – Instead of creating multiple subclasses for variations, we can wrap an object dynamically.
3. **Increases flexibility** – We can add multiple behaviors at runtime.

Example: Coffee Shop

Let's say we have a basic **coffee**, but we want to add **milk** or **sugar** as optional add-ons.

Step 1: Create a Base Interface

```
interface Coffee {
   String getDescription();
   double getCost();
}
```

✅ **Why?**

- This is the base type that all coffee types will follow.

Step 2: Create a Concrete Class (Base Component)

```java
class SimpleCoffee implements Coffee {
    public String getDescription() {
        return "Simple Coffee";
    }

    public double getCost() {
        return 5.0; // Base price
    }
}
```

✅ **Why?**

- This is the **original class** without extra features.

Step 3: Create an Abstract Decorator

```java
abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;

    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }

    public String getDescription() {
        return coffee.getDescription();
    }

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

✅ **Why?**

- It **wraps the base coffee object** and allows additional modifications.

Step 4: Create Concrete Decorators

```java
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    public String getDescription() {
        return super.getDescription() + ", Milk";
```

```java
    }

    public double getCost() {
        return super.getCost() + 1.5; // Extra cost for milk
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    public String getDescription() {
        return super.getDescription() + ", Sugar";
    }

    public double getCost() {
        return super.getCost() + 0.5; // Extra cost for sugar
    }
}
```

✅ **Why?**

- These classes **extend functionality dynamically** without modifying the SimpleCoffee class.

Step 5: Usage

```java
public class Main {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();
        System.out.println(coffee.getDescription() + " => $" + coffee.getCost());

        // Add Milk
        coffee = new MilkDecorator(coffee);
        System.out.println(coffee.getDescription() + " => $" + coffee.getCost());

        // Add Sugar
        coffee = new SugarDecorator(coffee);
        System.out.println(coffee.getDescription() + " => $" + coffee.getCost());
    }
}
```

✅ **Output:**

```
Simple Coffee => $5.0
Simple Coffee, Milk => $6.5
Simple Coffee, Milk, Sugar => $7.0
```

When to Use the Decorator Pattern?

- When you want to **extend an object's behavior dynamically**.
- When subclassing would create too many variations.
- When you need a **flexible and reusable solution**.