Design patterns are proven solutions to common problems faced during software design. They are templates designed to help developers write flexible, reusable, and efficient code.

**Why Learn Design Patterns?**

1. **Code Reusability**: Patterns provide reusable solutions.
2. **Improved Code Quality**: Helps in writing clean, maintainable code.
3. **Standardized Solutions**: Patterns are universally recognized, making it easier to collaborate with others.
4. **Problem-Solving**: Makes problem-solving faster and more systematic.

**Types of Design Patterns**

Design patterns are generally categorized into three types:

1. **Creational Patterns**: Deal with object creation.
2. **Structural Patterns**: Focus on class and object composition.
3. **Behavioral Patterns**: Deal with object interaction and responsibilities.

**1. Creational Patterns**

**Example: Singleton Pattern**

* Ensures a class has only one instance and provides a global access point to it.

**Code Example**:

class Singleton {

private static Singleton instance;

// Private constructor prevents instantiation from other classes

private Singleton() {}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

public class SingletonExample {

public static void main(String[] args) {

Singleton obj1 = Singleton.getInstance();

Singleton obj2 = Singleton.getInstance();

System.out.println(obj1 == obj2); // Output: true (same instance)

}

}

**1. Singleton Pattern**

* **Purpose**: Ensure only one instance of a class exists and provide a global access point to it.

**Code Explanation**:

java

Copy code

class Singleton {

private static Singleton instance; // Static variable to hold the single instance

private Singleton() {} // Private constructor to prevent instantiation

public static Singleton getInstance() {

if (instance == null) { // Check if instance is null

instance = new Singleton(); // Create instance if not already created

}

return instance; // Return the single instance

}

}

1. **Step 1**: Create a private static variable instance to hold the single instance.
2. **Step 2**: Make the constructor private to prevent creating objects using new Singleton().
3. **Step 3**: Provide a public static method getInstance() to return the single instance.
   * If instance is null, create the instance.
   * Otherwise, return the existing instance.

**Demo Explanation**:

java

Copy code

Singleton obj1 = Singleton.getInstance();

Singleton obj2 = Singleton.getInstance();

System.out.println(obj1 == obj2); // true

* Both obj1 and obj2 refer to the same instance because the Singleton pattern ensures only one object is created.

**2. Structural Patterns**

**Example: Adapter Pattern**

* Allows incompatible interfaces to work together by acting as a bridge.

**Code Example**:

interface USB {

void connectWithUSB();

class USBDevice implements USB {

public void connectWithUSB() {

System.out.println("Connected with USB.");

}

}

// Adapter to connect USB with HDMI

interface HDMI {

void connectWithHDMI();

}

class HDMIAdapter implements HDMI {

private USB usbDevice;

public HDMIAdapter(USB usbDevice) {

this.usbDevice = usbDevice;

}

public void connectWithHDMI() {

usbDevice.connectWithUSB();

System.out.println("Adapted to HDMI.");

}

}

public class AdapterExample {

public static void main(String[] args) {

USB usbDevice = new USBDevice();

HDMI hdmiAdapter = new HDMIAdapter(usbDevice);

hdmiAdapter.connectWithHDMI();

}

}

**Adapter Pattern**

* **Purpose**: Allow incompatible interfaces to work together by providing a wrapper (adapter) around the old interface.

**Code Explanation**:

java

Copy code

interface USB {

void connectWithUSB(); // Defines the USB interface

}

class USBDevice implements USB {

public void connectWithUSB() {

System.out.println("Connected with USB.");

}

}

interface HDMI {

void connectWithHDMI(); // Defines the HDMI interface

}

class HDMIAdapter implements HDMI {

private USB usbDevice; // Composition: Adapter wraps the USB device

public HDMIAdapter(USB usbDevice) {

this.usbDevice = usbDevice; // Initialize the adapter with the USB device

}

public void connectWithHDMI() {

usbDevice.connectWithUSB(); // Delegate the call to the USB device

System.out.println("Adapted to HDMI.");

}

}

1. **Step 1**: Define two incompatible interfaces, USB and HDMI.
2. **Step 2**: Implement the USBDevice class that works with the USB interface.
3. **Step 3**: Create an adapter class HDMIAdapter that implements the HDMI interface.
   * It wraps an instance of USBDevice and delegates the functionality.
4. **Step 4**: Call the connectWithHDMI() method on the adapter to adapt USB to HDMI.

**Demo Explanation**:

java

Copy code

USB usbDevice = new USBDevice();

HDMI hdmiAdapter = new HDMIAdapter(usbDevice);

hdmiAdapter.connectWithHDMI();

* The adapter enables a USB device to behave like an HDMI device.

**3. Behavioral Patterns**

**Example: Observer Pattern**

* Allows one object (subject) to notify multiple objects (observers) about state changes.

**Code Example**:

import java.util.ArrayList;

import java.util.List;

interface Observer {

void update(String message);

}

class User implements Observer {

private String name;

public User(String name) {

this.name = name;

}

public void update(String message) {

System.out.println(name + " received message: " + message);

}

}

class NotificationService {

private List<Observer> observers = new ArrayList<>();

public void addObserver(Observer observer) {

observers.add(observer);

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

public void notifyObservers(String message) {

for (Observer observer : observers) {

observer.update(message);

}

}

}

public class ObserverExample {

public static void main(String[] args) {

NotificationService service = new NotificationService();

Observer user1 = new User("Alice");

Observer user2 = new User("Bob");

service.addObserver(user1);

service.addObserver(user2);

service.notifyObservers("New update available!");

}

}

**3. Observer Pattern**

* **Purpose**: Notify multiple observers when the subject’s state changes.

**Code Explanation**:

java

Copy code

interface Observer {

void update(String message); // Observer's method to receive updates

}

class User implements Observer {

private String name; // Name of the observer (user)

public User(String name) {

this.name = name; // Initialize with the user's name

}

public void update(String message) {

System.out.println(name + " received message: " + message);

}

}

class NotificationService {

private List<Observer> observers = new ArrayList<>(); // List of observers

public void addObserver(Observer observer) {

observers.add(observer); // Add an observer to the list

}

public void removeObserver(Observer observer) {

observers.remove(observer); // Remove an observer from the list

}

public void notifyObservers(String message) {

for (Observer observer : observers) { // Notify all observers

observer.update(message);

}

}

}

1. **Step 1**: Define an Observer interface with an update() method.
2. **Step 2**: Implement the User class as a concrete observer.
   * Each user receives messages from the notification service.
3. **Step 3**: Implement the NotificationService class as the subject.
   * It maintains a list of observers and provides methods to add/remove them.
   * The notifyObservers() method sends updates to all registered observers.

**Demo Explanation**:

java

Copy code

NotificationService service = new NotificationService();

Observer user1 = new User("Alice");

Observer user2 = new User("Bob");

service.addObserver(user1);

service.addObserver(user2);

service.notifyObservers("New update available!");

* Alice and Bob are notified of the new update because they are registered observers.

**Best Practices When Using Design Patterns**

1. **Understand the Problem**: Use patterns only when they solve a real problem.
2. **Don’t Overuse**: Overcomplicating code with unnecessary patterns can make it hard to read and maintain.
3. **Combine Patterns**: Sometimes, combining patterns provides the best solution.
4. **Study Examples**: Understanding real-world use cases is the best way to learn patterns.

**Resources to Explore**

1. **Books**: *Head First Design Patterns* and *Design Patterns: Elements of Reusable Object-Oriented Software*.
2. **Practice**: Implement examples of each pattern to understand their applicability.

Here's a **step-by-step explanation for each design pattern example** mentioned above:

**2.**

**Key Points from Each Example**

**Singleton Pattern**

* Only one instance is created.
* Use private constructor and static method for controlled access.

**Adapter Pattern**

* Bridges two incompatible interfaces.
* The adapter uses composition to adapt one interface to another.

**Observer Pattern**

* Multiple objects (observers) listen to state changes in one object (subject).
* Helps implement the publisher-subscriber model.