Design Patterns in Java

1. Singleton Pattern

The Singleton pattern ensures that a class has only one instance and provides a global point of access to it. This is perfect for things like database connections or logger classes.

```
package Singleton;

public class DatabaseConfig {
    //1.make object static and private
    private static DatabaseConfig instance;

    //2.make constructor private so we cant create object with
constructor
    private DatabaseConfig(){}

    //3.create method to get instance of class
    public static DatabaseConfig getInstance(){
        if(instance == null){
            instance = new DatabaseConfig();
        }
        return instance;
    }

    //test method
    public void showMessage(){
        System.out.println("Connected to database");
    }

}
```

2. Factory Pattern (Creational)

The Factory pattern provides an interface for creating objects but allows subclasses to alter the type of objects that will be created. It hides the logic of instantiation from the user.

1.Notification.java

```
package FactoryPattern;

public interface Notification {
    void notifyUser();
}
```

2.EmailNotification.java

```
package FactoryPattern;

public class EmailNotification implements Notification{
    @Override
    public void notifyUser() {
        System.out.println("Email Notification Sent....");
    }
}
```

3.SMSNotification.java

```
package FactoryPattern;

public class SMSNotification implements Notification{
    @Override
    public void notifyUser() {
        System.out.println("SMS Notification Sent...");
    }
}
```

4.NotificationFactory.java

```
package FactoryPattern;

public class NotificationFactory {
    public static Notification createNotification(String type){
        if(type.equals("Email")) return new EmailNotification();
        if(type.equals("SMS")) return new SMSNotification();
        return null;
    }
}
```

5.NotificationApp.java

```
package FactoryPattern;

public class NotificationApp {
    public static void main(String[] args) {
        Notification notification =
NotificationFactory.createNotification("Email");
        notification.notifyUser();
        Notification notification1 =
NotificationFactory.createNotification("SMS");
```

```
        notification1.notifyUser();
    }
}
```

3. Builder Pattern

The Builder Pattern is perfect when you need to construct complex objects step-by-step, especially if some fields are optional. It helps you avoid the problem of constructor overloading, where you end up writing multiple constructors with different parameter combinations.

1.Computer.java

```
package BuilderDesignPattern;

public class Computer {
    //1.make every thing private
    private int hdd;
    private int ram;
    private boolean gpuEnabled;
    private boolean bluetoothEnabled;

    //2. make constructor private
    //so only builder can access it from inner class
    private Computer(ComputerBuilder builder){
        this.hdd = builder.hdd;
        this.ram = builder.ram;
        this.gpuEnabled = builder.gpuEnabled;
        this.bluetoothEnabled = builder.bluetoothEnabled;
    }

    //3. static inner builder class
    public static class ComputerBuilder{
        private int hdd;
        private int ram;
        private boolean gpuEnabled;
        private boolean bluetoothEnabled;

        public ComputerBuilder setHdd(int hdd) {
            this.hdd = hdd;
            return this;
        }

        public ComputerBuilder setRam(int ram) {
            this.ram = ram;
            return this;
        }
```

```
        public ComputerBuilder setGpuEnabled(boolean gpuEnabled)
{
            this.gpuEnabled = gpuEnabled;
            return this;
        }

        public ComputerBuilder setBluetoothEnabled(boolean
bluetoothEnabled) {
            this.bluetoothEnabled = bluetoothEnabled;
            return this;
        }

        public Computer build(){
            return new Computer(this);
        }
    }

    @Override
    public String toString() {
        return "Computer [ HDD = "+hdd+", RAM = "+ram+", GPU
Enabled = "+gpuEnabled+", Bluetooth Enabled =
"+bluetoothEnabled+" ]";
    }
}
```

2.ComputerApp.java

```
package BuilderDesignPattern;

public class ComputerApp {
    public static void main(String[] args) {
        Computer computer = new
Computer.ComputerBuilder().setHdd(500)
                .setRam(16)
                .setGpuEnabled(true)
                .setBluetoothEnabled(true)
                .build();

        Computer computer1 = new
Computer.ComputerBuilder().setHdd(1000)
                .setRam(12)
                .build();

        System.out.println(computer);
        System.out.println(computer1);
    }
}
```

4. Observer Pattern (Behavioral)

The Observer pattern defines a one-to-many dependency. When one object (the Subject) changes state, all its dependents (Observers) are notified and updated automatically. This is the heart of "event-driven" programming.

1.Observer.java

```
package ObserverDesignPattern;

public interface Observer {
    void update(String title);
}
```

2.Subscriber.java

```
package ObserverDesignPattern;

public class Subscriber implements Observer{
    private final String name;
    Subscriber(String name){
        this.name = name;
    }
    @Override
    public void update(String title) {
        System.out.println("Hey "+name+" new video uploaded
"+title);
    }
}
```

3.Channel.java

```
package ObserverDesignPattern;

import java.util.ArrayList;
import java.util.List;

public class Channel {
    List<Observer> subscribers = new ArrayList<>();

    //method to add subscriber when any one subscribe this
channel
    public void subscribe(Subscriber s){
```

```
        subscribers.add(s);
    }
    //upload method
    public void upload(String videoTitle){
        System.out.println(videoTitle+" uploaded!");
        //notify all subscribers
        for(Observer o : subscribers){
            o.update(videoTitle);
        }

    }

}
```

4.ObserverApp.java

```
package ObserverDesignPattern;

public class ObserverApp {
    public static void main(String[] args) {
        Channel ch = new Channel();
        Subscriber s1 = new Subscriber("Abc");
        Subscriber s2 = new Subscriber("Def");
        Subscriber s3 = new Subscriber("Ghe");
        Subscriber s4 = new Subscriber("Ijk");

        ch.subscribe(s1);
        ch.subscribe(s2);
        ch.subscribe(s3);
        ch.subscribe(s4);

        //till here channel ch have 4 subscriber s1,s2,s3,s4

        //calling upload method
        ch.upload("New Video");

    }
}
```

5. Strategy Pattern (Behavioral)

The Strategy Pattern allows you to define a family of algorithms, put each of them into a separate class, and make their objects interchangeable. This avoids messy if-else or switch blocks that grow every time you add a new feature.

1.PaymentStrategy.java

```
package StrategyDesignPattern;

public interface PaymentStrategy {
    void pay(int amount);
}
```

2.CreditCardPayment.java

```
package StrategyDesignPattern;

public class CreditCardPayment implements PaymentStrategy{
    @Override
    public void pay(int amount) {
        System.out.println("Paid "+amount+" using Credit Card");
    }
}
```

3.UPIPayment.java

```
package StrategyDesignPattern;

public class UPIPayment implements PaymentStrategy{
    @Override
    public void pay(int amount) {
        System.out.println("Paid "+amount+" using UPI payment");
    }
}
```

4.PaymentContext.java

```
package StrategyDesignPattern;

public class PaymentContext {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy
paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    //method to call pay()
    public void checkout(int amount){
        paymentStrategy.pay(amount);
```

```
    }
}
```

5.PaymentApp.java

```
package StrategyDesignPattern;

public class PaymentApp {
    public static void main(String[] args) {
        PaymentContext px = new PaymentContext();
        //setting strategy for credit card
        px.setPaymentStrategy(new CreditCardPayment());
        px.checkout(1000);
        //setting strategy for upi
        px.setPaymentStrategy(new UPIPayment());
        px.checkout(1000);
    }
}
```

6. Decorator Design Pattern

The Decorator pattern is a structural design pattern that allows behavior to be added to an individual object dynamically, without affecting the behavior of other instances of the same class.

1.Coffee.java

```
package DecoratorDesignPattern;

public interface Coffee {
    String getDescription();
    double getPrice();
}
```

2.CoffeeDecorator.java

```
package DecoratorDesignPattern;

 public abstract class CoffeeDecorator implements Coffee{
    protected Coffee decoratedCoffee;
    public CoffeeDecorator(Coffee coffee){
        this.decoratedCoffee = coffee;
    }
```

```
    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    @Override
    public double getPrice() {
        return decoratedCoffee.getPrice();
    }
}
```

3. MilkDecorator.java

```
package DecoratorDesignPattern;

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

    public String getDescription(){
        return decoratedCoffee.getDescription() + ",Milk";
    }

    public double getPrice(){
        return decoratedCoffee.getPrice()+1.5;
    }

}
```

4. SugarDecorator.java

```
package DecoratorDesignPattern;

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


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


    public double getPrice() {
```

```
        return decoratedCoffee.getPrice() + 0.5;
    }
}
```

5. CoffeeApp.java

```
package DecoratorDesignPattern;

public class CoffeeApp {
    public static void main(String[] args) {
        Coffee simpleCoffee = new SimpleCoffee();
        simpleCoffee = new MilkDecorator(simpleCoffee);
        simpleCoffee = new SugarDecorator(simpleCoffee);

        System.out.println(simpleCoffee.getDescription());
        System.out.println(simpleCoffee.getPrice());
    }
}
```

7. Prototype Pattern

The Prototype Pattern is a Creational pattern used when creating a new object from scratch is costly, complex, or time-consuming. Instead of building a new instance, you copy (clone) an existing one and modify it if necessary.

In Java, this is typically done by implementing the Cloneable interface and overriding the clone() method.

Shallow Copy vs. Deep Copy

This is a common interview question related to this pattern:

Shallow Copy: Only the "top-level" object is copied. If the object contains a List, both the original and the copy will point to the same List in memory.

Deep Copy: Every internal object and reference is copied. Changing the copy will never affect the original (This is what we did in the code above by creating a new ArrayList).

1.Employees.java

```
package PrototypeDesignPattern;

import java.util.ArrayList;
import java.util.List;

//1. implement cloneable interface
```

```
public class Employees implements Cloneable{

    List<String> empList;
    public Employees(){
        this.empList = new ArrayList<>();
    }

    public Employees(List<String> list){
        this.empList = list;
    }

    public List<String> getEmpList() {
        return empList;
    }

    //method which is taking time in creation
    public void loadData() {
        empList.add("Pankaj");
        empList.add("Raj");
        empList.add("David");
        empList.add("Lisa");
    }

    //2.override clone method
    @Override
    public Employees clone() throws CloneNotSupportedException{
        ArrayList<String> temp = new ArrayList<>();
        for(String emp : empList){
            temp.add(emp);
        }
        return new Employees(temp);
    }
}
```

2.Prototype.java

```
package PrototypeDesignPattern;

import java.util.List;

public class PrototypeApp {
    public static void main(String[] args) throws
CloneNotSupportedException{
        Employees emps = new Employees();
        //time consuming method called one time only
        emps.loadData();
```

```
        //create new emps object using clone
        Employees empNew = (Employees) emps.clone();
        //update list in second object
        List<String> empList = empNew.getEmpList();
        empList.add("John");

        //print both object list
        System.out.println(emps.getEmpList());
        System.out.println(empNew.getEmpList());
    }
}
```

8. Adapter Pattern

The Adapter Pattern is used when you want to make two incompatible interfaces work together. It acts like a bridge or a connector between two classes that otherwise cannot communicate.

1.JsonReader.java

```
package AdapterDesignPattern;

public interface JsonReader {
    void readJson(String json);
}
```

2.XmlService.java

```
package AdapterDesignPattern;

public class XMLService {
    public void readXml(String xmlData){
        System.out.println("Parsing xml data = "+xmlData);
    }
}
```

3.XmlToJsonAdapter.java

```
package AdapterDesignPattern;

public class XmlToJsonAdapter implements JsonReader{
    //1.legacy or third party class which need to inject
    private XMLService xmlService;

    public XmlToJsonAdapter() {
        this.xmlService = new XMLService();
    }
```

```
    @Override
    public void readJson(String json) {
        //logic to convert json data to xml(dummy
implementation)
        String convertedXmlData = "<xml>"+json+"</xml>";
        xmlService.readXml(convertedXmlData);
    }

}
```

4.AdapterApp.java

```
package AdapterDesignPattern;

public class AdapterApp {
    public static void main(String[] args) {
        String jsonData = "{'name':'MSY'}";
        JsonReader jsonReader = new XmlToJsonAdapter();
        jsonReader.readJson(jsonData);
    }
}
```

9. Proxy Pattern

The Proxy Pattern is a Structural pattern that provides a placeholder or "substitute" for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request reaches the original object.

1.Internet.java

```
package ProxyDesignPattern;

public interface Internet {
    void connectTo(String host);
}
```

2.RealInternet.java

```
package ProxyDesignPattern;

public class RealInternet implements Internet{
    @Override
```

```
    public void connectTo(String host) {
        System.out.println("Connecting to "+host);
    }
}
```

3.ProxyInternet.java

```
package ProxyDesignPattern;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class ProxyInternet implements Internet{
    private RealInternet realInternet;

    public ProxyInternet() {
        this.realInternet = new RealInternet();
    }
    //create list of banned site for access control
    public static List<String> bannedSites =
Arrays.asList("social.com","video.com","game.com");

    @Override
    public void connectTo(String host) {
        if (!bannedSites.contains(host)){
            realInternet.connectTo(host);
        }
        else {
            System.out.println("Sorry we can't connect");
        }
    }
}
```

4.ProxyApp.java

```
package ProxyDesignPattern;

public class ProxyApp {
    public static void main(String[] args) {
        Internet internet = new ProxyInternet();
        internet.connectTo("search.com");
        internet.connectTo("game.com");

    }

}
```

10. Command Design Pattern

The Command Pattern is a Behavioral pattern that turns a request into a stand-alone object. This object contains all the information about the request: the method to call, the arguments, and the object that performs the work.

1.TextFileOpeartion.java (Command)

```
//command
package CommandDesignPattern;
public interface TextFileOperation {
    void execute();
}
```

2.TextFile.java (Receiver)

```
//receiver
package CommandDesignPattern;

public class TextFile {
    private String fileName;
    public TextFile(String fileName) {
        this.fileName = fileName;
    }

    //receiver methods
    public void open(){
        System.out.println("Opening "+fileName);
    }
    public void save(){
        System.out.println("Saving "+fileName);
    }

}
```

3.OpenTextFileOperation.java (Concrete Implementation of Command)

```
//Concrete Commands
package CommandDesignPattern;

public class OpenTextFileOperation implements TextFileOperation{
    private final TextFile textFile;

    public OpenTextFileOperation(TextFile textFile) {
        this.textFile = textFile;
```

```
    }

    @Override
    public void execute() {
        textFile.open();
    }
}
```

4.SaveTextFileOperation.java (Concrete Implementation of Command)

```
//Concrete Commands
package CommandDesignPattern;

public class SaveTextFileOperation implements TextFileOperation{
    private final TextFile textFile;

    public SaveTextFileOperation(TextFile textFile) {
        this.textFile = textFile;
    }

    @Override
    public void execute() {
        textFile.save();
    }
}
```

5.TextFileOprationExecutor.java (Invoker)

```
//invoker
package CommandDesignPattern;

import java.util.ArrayList;
import java.util.List;

public class TextFileOperationExecutor {
    private List<TextFileOperation> opsList;
    TextFileOperation textFileOperation;

    public TextFileOperationExecutor() {
        this.opsList = new ArrayList<>();
    }

    public void setTextFileOperation(TextFileOperation
textFileOperation){
        this.textFileOperation = textFileOperation;
        opsList.add(textFileOperation);
    }
```

```
    public void executeOperation(){
        textFileOperation.execute();
    }
}
```

6.CommandApp.java

```
package CommandDesignPattern;

public class CommandApp {
    public static void main(String[] args) {
        TextFileOperationExecutor invoker = new
TextFileOperationExecutor();

        TextFile textFile = new TextFile("file1.txt");
        OpenTextFileOperation openTextFileOperation = new
OpenTextFileOperation(textFile);
        //sets open file ops
        invoker.setTextFileOperation(openTextFileOperation);
        invoker.executeOperation();

        //save file ops
        SaveTextFileOperation saveTextFileOperation = new
SaveTextFileOperation(textFile);
        invoker.setTextFileOperation(saveTextFileOperation);
        invoker.executeOperation();
    }
}
```