**Top 30 Java Design Patterns Interview Questions**

**Q1: What is a design pattern in Java?**

A design pattern refers to an established and verified solution for a repetitive issue in software development. It offers a standardized approach to solve common design problems and helps developers write maintainable, extensible, and reusable code.

**Q2: Difference between design pattern and architectural pattern.**

* **Design patterns** are like recipes for solving problems in software development. They are helpful because they provide ready-made and flexible solutions to common problems that software developers face. These solutions make the software better. Design patterns are focused on fixing problems in a specific type of software design, like how objects are organized and used.
* **Architectural patterns** help us to organize how an application is built. They give us a framework for how everything fits together. They decide what things a big program is made of and how they work together. These patterns help us to make sure that our program is easy to change and can grow to handle lots of users. They also help us to keep our program safe.

**Q3: What are the types of design patterns in Java?**

There are four types of design patterns in Java:

* Creational
* Structural
* Behavioral
* J2EE

**Q4: What is the Singleton design pattern in Java?**

The singleton design pattern is a way to make sure that only one copy of a class exists. It is used when we want to have a global way to access this class. We use the singleton pattern when we need one object to control things in the whole system.

To implement the Singleton Pattern, we will need to:

* Define a class with a private constructor, so that no other class can instantiate it.
* Declare a static variable of the same type as the class and create an instance of the class in the variable.
* Declare a static method that returns the instance of the class.

|  |
| --- |
| public class Singleton {      private static Singleton instance = null;      private Singleton() {          // private constructor      }      public static synchronized Singleton getInstance() {          if (instance == null) {              instance = new Singleton();          }          return instance;      }  } |

**Q5: What is the Factory Method design pattern in Java?**

The Factory Method pattern is a way to make things without saying exactly what kind of thing to make. It makes an outline for making things but let’s different kinds of things choose what kind of thing to make. This is helpful when you want to make things without worrying about how they are made.

To implement the Factory Pattern, we will need to:

* Define an interface or abstract class that declares the factory method for creating objects.
* Create concrete classes that implement the interface or abstract class and provide their own implementation of the factory method.
* Define a client class that uses the factory method to create objects without knowing their exact class.

|  |
| --- |
| public interface Animal {      void speak();  }    public class Dog implements Animal {      @Override public void speak()      {          System.out.println("Dog says: Bow Wow!");      }  }    public class Cat implements Animal {      @Override public void speak()      {          System.out.println("Cat says: Meow!");      }  }    public interface AnimalFactory {      Animal createAnimal();  }    public class DogFactory implements AnimalFactory {      @Override public Animal createAnimal()      {          return new Dog();      }  }    public class CatFactory implements AnimalFactory {      @Override public Animal createAnimal()      {          return new Cat();      }  }    public class AnimalClient {      public static void main(String[] args)      {          AnimalFactory dogFactory = new DogFactory();          Animal dog = dogFactory.createAnimal();          dog.speak();          AnimalFactory catFactory = new CatFactory();          Animal cat = catFactory.createAnimal();          cat.speak();      }  } |

**Q6: What is the Observer design pattern in Java?**

The Observer pattern is a way for objects to talk to each other. It’s like when you tell your friends about something, and they all know what’s going on. It’s used when many objects need to know when something happens to one object.

To implement the Observer Pattern, we will need to:

* Define an interface for the Observer objects that specifies the method(s) to be called when the state of the observed object changes.
* Define an interface for the Subject (observed) object that allows Observer objects to register, remove, and notify them about changes.
* Create concrete Observer classes that implement the Observer interface and provide their own implementation of the update method.
* Create a concrete Subject class that implements the Subject interface and maintains a list of registered Observer objects. This class also provides methods to modify the state of the object, which triggers notifications to all registered Observers.

|  |
| --- |
| // Observer interface  interface Observer {      void update(int value);  }    // Subject interface  interface Subject {      void registerObserver(Observer observer);      void removeObserver(Observer observer);      void notifyObservers();  }    // Concrete Observer class  class ConcreteObserver implements Observer {      @Override      public void update(int value) {          System.out.println("Value changed to " + value);      }  }    // Concrete Subject class  class ConcreteSubject implements Subject {      private List<Observer> observers = new ArrayList<>();      private int value;        public void setValue(int value) {          this.value = value;          notifyObservers();      }        @Override      public void registerObserver(Observer observer) {          observers.add(observer);      }        @Override      public void removeObserver(Observer observer) {          observers.remove(observer);      }        @Override      public void notifyObservers() {          for (Observer observer : observers) {              observer.update(value);          }      }  }    // Client code  public class ObserverDemo {      public static void main(String[] args) {          ConcreteSubject subject = new ConcreteSubject();            // Create observers and register them          Observer observer1 = new ConcreteObserver();          subject.registerObserver(observer1);            Observer observer2 = new ConcreteObserver();          subject.registerObserver(observer2);            // Modify the value of the subject, which triggers notifications to all observers          subject.setValue(10);            // Remove one observer and modify the value again          subject.removeObserver(observer1);          subject.setValue(20);      }  } |

**Q7: What is the Adapter design pattern in Java?**

The Adapter pattern helps different things to work together, even if they are not originally made to work together. It changes one thing to be like another thing so that they can work together. This is useful when there are things that cannot work together, but we still need them to work together.

To implement the Adapter Pattern, we will need to:

* Identify the target interface that the client expects to use.
* Identify the existing interface that needs to be adapted to meet the target interface.
* Create an Adapter class that implements the target interface and contains an instance of the existing interface.
* Implement the methods of the target interface in the Adapter class by calling the appropriate methods of the existing interface instance.

|  |
| --- |
| // Target interface  interface MediaPlayer {      void play(String audioType, String fileName);  }    // Existing interface  interface AdvancedMediaPlayer {      void playVlc(String fileName);      void playMp4(String fileName);  }    // Existing class that implements the AdvancedMediaPlayer interface  class VlcPlayer implements AdvancedMediaPlayer {      @Override      public void playVlc(String fileName) {          System.out.println("Playing vlc file. Name: " + fileName);      }        @Override      public void playMp4(String fileName) {          // do nothing      }  }    // Existing class that implements the AdvancedMediaPlayer interface  class Mp4Player implements AdvancedMediaPlayer {      @Override      public void playVlc(String fileName) {          // do nothing      }        @Override      public void playMp4(String fileName) {          System.out.println("Playing mp4 file. Name: " + fileName);      }  }    // Adapter class that implements the MediaPlayer interface and contains an instance of the AdvancedMediaPlayer interface  class MediaAdapter implements MediaPlayer {      private AdvancedMediaPlayer advancedMediaPlayer;        public MediaAdapter(String audioType) {          if (audioType.equalsIgnoreCase("vlc")) {              advancedMediaPlayer = new VlcPlayer();          } else if (audioType.equalsIgnoreCase("mp4")) {              advancedMediaPlayer = new Mp4Player();          }      }        @Override      public void play(String audioType, String fileName) {          if (audioType.equalsIgnoreCase("vlc")) {              advancedMediaPlayer.playVlc(fileName);          } else if (audioType.equalsIgnoreCase("mp4")) {              advancedMediaPlayer.playMp4(fileName);          }      }  }    // Client code  public class AdapterDemo {      public static void main(String[] args) {          MediaPlayer mediaPlayer = new MediaAdapter("mp4");          mediaPlayer.play("mp4", "movie.mp4");            mediaPlayer = new MediaAdapter("vlc");          mediaPlayer.play("vlc", "song.vlc");      }  } |

**Q8: What is the Decorator design pattern in Java?**

The Decorator pattern is a way to add new things to an object without changing how it looks or works. You make a new class that goes around the old object and adds new stuff to it.

To implement the Decorator Pattern, we will need to:

* Create an interface or abstract class that defines the methods to be implemented by both the original object and the decorator.
* Create a concrete class that implements the interface or abstract class and represents the original object.
* Create a decorator class that implements the same interface or abstract class and contains an instance of the original object.
* Implement the methods of the interface or abstract class in the decorator class by calling the appropriate methods of the original object instance and adding new functionality.

|  |
| --- |
| // Interface or abstract class  interface Car {      void assemble();  }    // Concrete class that implements the Car interface  class BasicCar implements Car {      @Override      public void assemble() {          System.out.println("Basic Car.");      }  }    // Decorator class that implements the Car interface and contains an instance of the Car interface  class CarDecorator implements Car {      protected Car car;        public CarDecorator(Car car) {          this.car = car;      }        @Override      public void assemble() {          car.assemble();      }  }    // Concrete decorator class that adds new functionality to the original object  class SportsCarDecorator extends CarDecorator {      public SportsCarDecorator(Car car) {          super(car);      }        @Override      public void assemble() {          super.assemble();          System.out.println("Adding features of Sports Car.");      }  }    // Client code  public class DecoratorDemo {      public static void main(String[] args) {          Car basicCar = new BasicCar();          basicCar.assemble();            Car sportsCar = new SportsCarDecorator(new BasicCar());          sportsCar.assemble();      }  } |

**Q9: What is the Command design pattern in Java?**

The Command pattern is a way to organize and control requests in a computer program. Instead of directly giving instructions, we use objects called commands to represent and execute our requests. This helps us use different requests with different clients, keep a record of previous requests, and even undo previous actions. We create an interface or a class to define how commands should be executed, and then create different classes to represent different types of commands.

To implement the Command Pattern, we will need to:

* Create a Command interface or abstract class that defines the execute method.
* Create concrete classes that implement the Command interface and represent different commands. These classes should contain a reference to the receiver object that will execute the command.
* Create an invoker class that will execute the commands by calling their execute method.
* Client code should create instances of the concrete Command classes and pass them to the invoker class.

|  |
| --- |
| // Command interface  interface Command {      void execute();  }    // Concrete command classes  class LightOnCommand implements Command {      private Light light;        public LightOnCommand(Light light) {          this.light = light;      }        @Override      public void execute() {          light.turnOn();      }  }    class LightOffCommand implements Command {      private Light light;        public LightOffCommand(Light light) {          this.light = light;      }        @Override      public void execute() {          light.turnOff();      }  }    // Receiver class  class Light {      public void turnOn() {          System.out.println("Light turned on");      }        public void turnOff() {          System.out.println("Light turned off");      }  }    // Invoker class  class RemoteControl {      private Command command;        public void setCommand(Command command) {          this.command = command;      }        public void pressButton() {          command.execute();      }  }    // Client code  public class CommandDemo {      public static void main(String[] args) {          Light light = new Light();          Command lightOn = new LightOnCommand(light);          Command lightOff = new LightOffCommand(light);            RemoteControl remoteControl = new RemoteControl();          remoteControl.setCommand(lightOn);          remoteControl.pressButton();            remoteControl.setCommand(lightOff);          remoteControl.pressButton();      }  } |

**Q10: What is the Iterator design pattern in Java?**

The Iterator pattern helps us go through a list or a group of things. It keeps the way the group is organized a secret and gives us a way to get to each thing one after the other. It does this by creating a special tool called an Iterator that helps us move through the group.

To implement the Iterator Pattern, we will need to:

* Create an Iterator interface or abstract class that defines methods for accessing the elements of the aggregate object. Typically, these methods include hasNext() to check if there are more elements, and next() to return the next element.
* Create concrete classes that implement the Iterator interface and represent different types of iterators. These classes should contain a reference to the aggregate object they are iterating over.
* Create a concrete aggregate class that implements an Iterable interface or defines a method that returns an Iterator object. This method should create and return an instance of the appropriate Iterator class.
* Client code can use the Iterator to access the elements of the aggregate object.

|  |
| --- |
| // Iterator interface  interface Iterator<T> {      boolean hasNext();      T next();  }    // Concrete iterator class  class ArrayIterator<T> implements Iterator<T> {      private T[] array;      private int currentIndex;        public ArrayIterator(T[] array) {          this.array = array;          this.currentIndex = 0;      }        public boolean hasNext() {          return currentIndex < array.length;      }        public T next() {          return array[currentIndex++];      }  }    // Aggregate class  class Array<T> implements Iterable<T> {      private T[] array;        public Array(T[] array) {          this.array = array;      }        public Iterator<T> iterator() {          return new ArrayIterator<T>(array);      }  }    // Client code  public class IteratorDemo {      public static void main(String[] args) {          Integer[] array = {1, 2, 3, 4, 5};          Array<Integer> integerArray = new Array<Integer>(array);            Iterator<Integer> iterator = integerArray.iterator();          while (iterator.hasNext()) {              Integer element = iterator.next();              System.out.println(element);          }      }  } |

**Q11: What is the Template Method design pattern in Java?**

The Template Method pattern is a way to design things that lets you make a list of steps to do something, but some of the steps can be done in different ways. You start by making a base list of steps, then you make a main step that tells the other steps to run. Each step can be different, and they all work together to do the thing you want to do.

To implement the Template Pattern, we will need to:

* Create an abstract base class that defines a template method. The template method should call other methods (abstract or concrete) to perform specific steps of the algorithm.
* Define abstract methods in the base class for the steps that need to be implemented by subclasses.
* Create concrete subclasses that implement the abstract methods to provide specific implementations for the steps.
* Client code can create an instance of the concrete subclass and call the template method to execute the algorithm.

|  |
| --- |
| // Abstract base class with a template method  abstract class AbstractClass {      public final void templateMethod() {          step1();          step2();          step3();      }        protected abstract void step1();      protected abstract void step2();      protected abstract void step3();  }    // Concrete subclass that provides specific implementations for the steps  class ConcreteClass extends AbstractClass {      protected void step1() {          System.out.println("ConcreteClass: step 1");      }        protected void step2() {          System.out.println("ConcreteClass: step 2");      }        protected void step3() {          System.out.println("ConcreteClass: step 3");      }  }    // Client code  public class TemplateMethodDemo {      public static void main(String[] args) {          AbstractClass abstractClass = new ConcreteClass();          abstractClass.templateMethod();      }  } |

**Q12: What is the Builder Method design pattern in Java?**

The Builder design pattern is a way to build things step-by-step. It helps you make complicated things using a specific process. This pattern separates the building part from the final thing, so you can make different final things using the same process. The Builder pattern is useful when you want to make complicated things, but the process to make them is simple. It lets you make the thing little by little and then get the final thing when it’s done.

Here is an example of how the Builder Pattern might be implemented:

|  |
| --- |
| public class BuilderExample {      public static void main(String[] args) {          // create the director          Director director = new Director();            // create the builder          Builder builder = new ConcreteBuilder();            // construct the complex object          director.construct(builder);            // retrieve the finished product          ComplexObject complexObject = builder.getResult();      }  }    class ComplexObject {      // fields for the complex object  }    class Director {      public void construct(Builder builder) {          // use the builder to construct the complex object      }  }    abstract class Builder {      // methods for constructing the complex object      public abstract ComplexObject getResult();  }    class ConcreteBuilder extends Builder {      private ComplexObject object = new ComplexObject();        // methods for constructing the complex object      public ComplexObject getResult() {          return object;      }  } |

**Q13: How many ways we you create a singleton pattern?**

There are two ways of creating a Singleton pattern.

1. **Early Instantiation:**It is responsible for the creation of instance at load time.
2. **Lazy Instantiation:** It is responsible for the creation of instance when required.

**Q14: Describe the strategy to design a pattern?**

The following points should need to be taken care to describe the design pattern:

* The Pattern name and classification.
* The Problem and solution.
* **Consequences:** Variation and language-dependent alternatives should also be addressed.
* **Uses:** Identify the uses in the real systems and its efficiency.

**Q15: What is the decorator pattern in Java explain it with an example?**

The decorator pattern is one of the popular Java design patterns. It is common because of its heavy usage in java.io (package). The Decorator Pattern uses composition in place of inheritance to extend the functionality of an object at runtime.

Example: **BufferedReader** and **BufferedWriter**

**Q16: Difference between Strategy and State design Pattern in Java?**

Both Strategy and State patterns have a similar structure. Their UML class diagram looks the same, but they have different purposes. The state design pattern is used to manage and define the state of an object, while the Strategy pattern is used to describe a set of an interchangeable algorithm.

**Q17: What are the advantages of Composite Design Pattern in Java?**

The composite design pattern helps people work together with things that might or might not be part of a group.

**Advantages:**

* It describes the class hierarchy that contains primitive and complex objects.
* It makes it easy to add new kinds of the component.
* It facilitates with the flexibility of structure with a manageable class or interface.

**Q18: Describe the uses of the Composite Design Pattern.**

**Use cases:**

* When we want to represent a partial or full hierarchy of objects.
* In case we need to add the responsibilities dynamically to the individual object without affecting other objects.

**Q19: What are Some Design Patterns used in the JDK library?**

Some of the following design patterns are used in the JDK library:

* The decorator pattern is used by Wrapper classes.
* Singleton pattern is used by Calendar classes (Runtime).
* The Wrapper classes use factory pattern like Integer.valueOf.
* Event handling frameworks use observer pattern like swing, Abstract Window Toolkit (AWT).

**Q20: Advantage of Builder design pattern in Java.**

* **Immutable Objects:** Builders can enforce immutability by setting the object’s fields during construction and providing no way to modify them afterward. Immutable objects are thread-safe by nature, which can help avoid concurrency issues.
* It facilitates with a clear separation between the construction and representation of an object.
* It provides improved control over the construction process.
* **Named Parameters:** Builders allow you to use named parameters, which enhance code readability by making it clear which value corresponds to which field.
* In design Pattern, the object is always instantiated in a complete state.

**Q21: How to write Thread-safe Singleton in Java?**

* By writing singleton using double-checked locking.
* By using static Singleton instance initialized during class loading.
* By using Java enum to create a thread-safe singleton, this is the most straightforward way.

**Q22: Is it possible to create a clone of a singleton object?**

Yes, it is possible to create a clone of a singelton object

**Q23: What is the proxy pattern, and what does it do?**

It is also known as placeholder or surrogates. The term Proxy stands for an object representing another object. The proxy pattern provides a substitute or placeholder for another purpose to control access to it. According to Gangs of four (GOF), a Proxy Pattern “provides control for accessing the original object. “We can perform many security operations like hiding the information of the original object, on-demand loading, etc.

**Q24: Explain some different type of proxies?**

**Protection proxy**

It controls access to the real subject based on the condition.

**Virtual proxies**

It is a structural design pattern that falls under the category of proxy patterns. A proxy is an object that acts as an intermediary or placeholder for another object to control access to it.

**Caching proxies**

It is a type of proxy pattern that is used to improve the performance and efficiency of an application by storing and reusing previously fetched or computed data. It acts as an intermediary between a client and a target object, and its primary purpose is to cache data or results of expensive operations so that subsequent requests for the same data can be satisfied quickly from the cache rather than going to the target object or performing the computation again. Caching proxies are commonly used in various software systems, including web applications, databases, and distributed systems, to reduce latency and resource consumption.

**Remote proxies**

Remote proxies are used in distributed object communication. The remote proxy causes execution on the remote object by invoking a local object method.

**Smart proxies**

Smart proxies are used to implement log calls and reference counting to the object.

**Q25: Explain the Chain of Responsibility Pattern, when it is used & their advantages.**

It is a behavioral design pattern that allows you to pass requests along a chain of handlers. Each handler in the chain decides either to process the request or to pass it to the next handler in the chain. This pattern promotes loose coupling between the sender (client) of a request and its receivers (handlers), making it easier to add or remove handlers and change the order in which they process requests.

Advantages of the Chain of Responsibility pattern is:

* **Reduced Coupling:**The pattern promotes loose coupling between the sender and receivers of a request. The client doesn’t need to know which handler will process the request, and handlers don’t need to know about the client or other handlers. This separation of concerns makes the system more maintainable and flexible.
* **Dynamic Handling:** You can easily add, remove, or reorder handlers in the chain without affecting the client code. This dynamic configuration makes it easy to adapt the system to changing requirements.
* **Single Responsibility Principle:** Each handler has a single responsibility: to process a specific type of request or condition. This adheres to the Single Responsibility Principle and makes it easier to understand and maintain each handler.
* **Fallback Mechanism:** Handlers can provide a fallback mechanism. If a handler cannot process a request, it can pass it to the next handler in the chain, ensuring that the request is eventually handled or that a default action is taken.

**Uses** of Chain of Responsibility Pattern:

* When more than one objects are ready to handle a request, and the handler is unknown.
* In case the collection or a group of objects that can handle the request must be specified dynamically.

**Q26: How Bridge Pattern is different from the Adapter Pattern?**

**Bridge Pattern** focuses on separating abstraction and implementation hierarchies to allow them to evolve independently. It’s used to handle variations in both abstraction and implementation.

**Adapter Pattern** focuses on making two incompatible interfaces work together by providing a wrapper (adapter) that translates between them. It’s used to make existing classes collaborate when their interfaces don’t match.

**Q27: What is the difference between the Dependency Injection and Service Locator Patterns?**

**Dependency Injection (DI)** is a design pattern focused on managing and injecting dependencies into classes, promoting testability and loose coupling. It provides explicit and constructor-based or setter-based injection of dependencies.

**Service Locator (SL)** is a design pattern focused on centralizing the retrieval of services or components from a registry or locator object. It provides a centralized point for service location but doesn’t necessarily promote the same level of inversion of control or testability as Dependency Injection.

**Q28: Explain the Intercepting Filter Design Pattern, components and its advantages?**

The Intercepting Filter Design Pattern is a behavioral design pattern that allows you to preprocess and post-process requests and responses in a system by using a series of filters. Each filter performs a specific action or validation on the request or response as it flows through the system. This pattern is commonly used in web applications for tasks such as input validation, authentication, logging, and other cross-cutting concerns.

**Components of Intercepting Filter Pattern are:**

* **Filter:** Filters are responsible for processing requests and responses. They implement a common interface or extend a base filter class. Each filter performs a specific task, such as input validation, authentication, logging, or authorization.
* **Filter Chain:** The filter chain is a sequence of filters arranged in a specific order. When a request enters the system, it passes through each filter in the chain in order. The filter chain manages the order of execution and ensures that each filter is applied as needed.
* **Target:** The target represents the core functionality of the system. It is the component that handles the actual business logic or processing of requests. Filters are used to preprocess and post-process requests and responses before and after they reach the target.
* **Client:** The client initiates the request and specifies the filter chain to be applied. It communicates with the target indirectly through the filter chain.

**Advantages of Intercepting Filter Design Pattern:**

* Filter pattern provides central control with loosely coupled handlers.
* It expands reusability.
* The new filter can be added at any time without affecting the client’s code.
* Filters can be selected dynamically during program execution.

**Q29: Explain Data Access Object (DAO) Design Pattern?**

The Data Access Object (DAO) Design Pattern is a structural pattern that provides an abstract interface for accessing and manipulating data stored in a database or other data storage systems. The primary goal of the DAO pattern is to separate the business logic from the data access logic, promoting a clean and maintainable architecture. It accomplishes this by encapsulating the data access code within a set of data access objects, thereby abstracting the details of how data is fetched or persisted.

**Q30: what is the difference between Value Object (VO) and Java Data Object (JDO)?**

**VOs** are a design pattern for representing data as immutable objects without identity, represents an abstract design pattern, which is used in conjunction with entity beans, JDBC and JDO.

**JDOs** is a technology and specification for persisting Java objects in a database, persistent technology that competes against entity beans. It allows to create POJO (plain old java objects) and persevere them to the database.