***SYSTEM DESIGN –NS LONI***

Topics:

1. Introduction to LLD
2. History of Programming
3. Pillars of OOPS
4. UML Diagram
5. SOLID Design Principles
6. Project-1 [Build Google Docs]
7. Strategy Design Pattern
8. Factory Design Pattern
9. Factory Method
10. Abstract Factory Method
11. Singleton Design Pattern
12. As
13. As
14. As
15. As
16. As
17. As
18. As
19. As
20. As

**Topic 1 : Introduction to LLD**

**What is LLD**

**LLD Involves**

**DSA v/s System Design**

**LLD v/s HLD**

**Overview of DSA, LLD, HLD**

Here is a detailed, **interview-ready explanation** of **“Introduction to LLD (Low-Level Design)”**, including **real-world examples**, **differences with DSA & HLD**, and **top interview questions with answers** for FAANG+ and product-based companies:

**Topic 1: Introduction to LLD (Low-Level Design)**

**What is LLD?**

**Low-Level Design (LLD)** is the process of designing the **detailed internal structure** of a software module or system after the High-Level Design (HLD) is complete.

It deals with **class diagrams**, **design patterns**, **interfaces**, **object interactions**, **database schema**, and **business logic implementation**.

**Key Points:**

* LLD focuses on **how** each component behaves internally.
* It is **code-level design** that often leads directly to implementation.
* Includes **SOLID principles**, **OOP**, **Design Patterns** (like Singleton, Factory, Observer), and **UML class diagrams**.

**LLD Involves:**

| **Component** | **Description** |
| --- | --- |
| **OOPs concepts** | Classes, inheritance, polymorphism, abstraction, encapsulation |
| **Design Principles** | SOLID, DRY, KISS, YAGNI |
| **Design Patterns** | Singleton, Strategy, Factory, Builder, Observer, etc. |
| **UML Diagrams** | Class, Sequence, Activity diagrams |
| **Modularization** | Dividing systems into well-defined modules |
| **Code Extensibility** | Code that can evolve without breaking |

**DSA vs System Design**

| **Aspect** | **DSA** | **System Design** |
| --- | --- | --- |
| Focus | Algorithms & data structures | || Architecture & component interactions |
| Examples | Sorting, Graphs, Trees | || Building scalable systems like Uber, YouTube |
| Scope | Individual problem solving | || End-to-end application |
| Interview Round | Initial technical screening | || Mid to final rounds (design focused) |

**LLD vs HLD**

| **Feature** | **HLD (High-Level Design)** | **LLD (Low-Level Design)** |
| --- | --- | --- |
| Focus | Architecture, modules, services | Classes, interfaces, logic |
| Abstraction Level | High | Low |
| Includes | System architecture, APIs, databases | Class diagrams, design patterns, logic |
| Example | Designing Uber’s architecture (service components) | Designing Rider, Trip, Payment class & logic |
| Tools | Block diagrams, service diagrams | UML class, sequence diagrams |

**Overview: DSA, LLD, HLD**

| **Topic** | **Scope** | **Goal** | **Usage in Interview** |
| --- | --- | --- | --- |
| **DSA** | Logic-based problems | Optimize performance | Online rounds, first tech rounds |
| **LLD** | Internal system design | Clean & maintainable code | Mid tech rounds |
| **HLD** | Macro system design | Scalability, availability | Design rounds or system design rounds |

**Real-Life Example: BookMyShow**

* **HLD**: Microservices like UserService, MovieService, BookingService, PaymentService.
* **LLD**:
  + Create User, Theatre, Seat, Booking, and Payment classes.
  + Use Strategy Pattern for different payment methods.
  + Use Observer Pattern for sending confirmation notifications.

**Interview-Ready Question Bank (with Answers)**

**Q1: What is Low-Level Design, and why is it important?**

**Answer**:  
Low-Level Design (LLD) refers to the detailed design of components and classes. It ensures that the system’s implementation is **modular, scalable, reusable**, and **maintainable**. LLD bridges the gap between high-level architecture and actual coding.

**Q2: What are the key principles followed in LLD?**

**Answer**:  
Key principles include:

* **SOLID principles**:
  + **S**: Single Responsibility
  + **O**: Open/Closed
  + **L**: Liskov Substitution
  + **I**: Interface Segregation
  + **D**: Dependency Inversion
* **DRY** (Don’t Repeat Yourself)
* **KISS** (Keep It Simple, Stupid)
* **YAGNI** (You Ain’t Gonna Need It)

**Q3: How is LLD different from HLD?**

**Answer**:  
HLD defines the **macro architecture**: services, databases, modules, and APIs.  
LLD defines the **micro structure**: classes, methods, interfaces, design patterns.

* HLD → Blueprint of the house
* LLD → Interior design and wiring

**Q4: What are some *design patterns* used in LLD?**

**Answer**:

* **Singleton**: Ensures a class has only one instance.
* **Factory**: Used to create objects without exposing creation logic.
* **Observer**: Notify objects when state changes (e.g., Notification system).
* **Strategy**: Enables selecting algorithms at runtime (e.g., Payment method).
* **Builder**: For constructing complex objects (e.g., building a Resume object).

**Q5: What tools and diagrams are used in LLD?**

**Answer**:

* **UML Class Diagrams** – for class structure
* **Sequence Diagrams** – for interaction flow
* **Activity Diagrams** – for workflows
* **State Diagrams** – for state transitions

Tools: Draw.io, Lucidchart, StarUML

**Q6: What are some challenges in LLD?**

**Answer**:

* Handling future extensibility
* Avoiding tight coupling
* Managing dependencies
* Designing for reusability
* Ensuring testability

**Q7: Give an example of using a Design Pattern in LLD.**

**Answer**:  
In a **Food Delivery App**, different delivery strategies (bike, car, drone) can be implemented using the **Strategy Pattern**.  
Define an interface DeliveryStrategy, and classes like BikeDelivery, CarDelivery implement it.  
At runtime, choose the strategy based on order type or distance.

**Summary Table for Quick Revision**

| **Aspect** | **LLD** |
| --- | --- |
| Focus | Class-level and object-level design |
| Based On | OOP, SOLID, Design Patterns |
| Output | Class diagrams, sequence diagrams, design patterns |
| Objective | Maintainable, extensible codebase |
| Tools | UML tools (Lucidchart, Draw.io), IDE |
| Common Qs | SOLID, LLD vs HLD, design patterns, real-world scenarios |

**Further Learning Resources**

1. **Grokking the Object-Oriented Design Interview** (educative.io)
2. **System Design Primer** (GitHub)
3. YouTube: Gaurav Sen’s LLD videos
4. Book: “Design Patterns: Elements of Reusable Object-Oriented Software”

Here's a **diagram-based LLD example** for a **Ride-Sharing Application** like Uber or Ola. This example will cover:

1. Requirements
2. Class Diagram (UML)
3. Design Patterns Used
4. SOLID Principles Application
5. Interview Questions with Answers based on this LLD

**Low-Level Design Example: Ride-Sharing Application**

**Step 1: Requirements (Functional Scope)**

Design a basic ride-sharing system that allows:

* Users to register and book rides
* Drivers to register and accept rides
* Matching between riders and nearby available drivers
* Trip tracking and fare calculation
* Payment processing

**Step 2: Identify Core Entities**

* **User**
* **Driver**
* **Ride**
* **Vehicle**
* **Payment**
* **Location**
* **RideService**
* **RideMatchingService**
* **NotificationService**

**Step 3: UML Class Diagram (Textual Representation)**

+--------------------+

| **User** |

+--------------------+

| +userId |

| +name |

| +phone |

+--------------------+

| +bookRide() |

+--------------------+

+--------------------+

| **Driver** |

+--------------------+

| +driverId |

| +name |

| +vehicle |

| +location |

| +isAvailable |

+--------------------+

| +acceptRide() |

+--------------------+

+--------------------+

| **Vehicle** |

+--------------------+

| +vehicleNo |

| +type |

| +model |

+--------------------+

+--------------------+

| **Ride** |

+--------------------+

| +rideId |

| +user |

| +driver |

| +source |

| +destination |

| +fare |

| +status |

+--------------------+

| +startRide() |

| +endRide() |

+--------------------+

+--------------------------+

| **RideMatchingService** |

+--------------------------+

| +matchDriver(user) |

+--------------------------+

+------------------------+

| **PaymentService** |

+------------------------+

| +calculateFare() |

| +processPayment() |

+------------------------+

+--------------------------+

| **NotificationService** |

+--------------------------+

| +sendRideConfirmation() |

| +sendETA() |

+--------------------------+

**Step 4: Design Patterns Used**

| **Pattern** | **Where it's used** | **Purpose** |
| --- | --- | --- |
| **Strategy Pattern** | PaymentService | Allows multiple payment methods like UPI, Card, Cash |
| **Singleton Pattern** | RideMatchingService | Only one instance to manage matching |
| **Observer Pattern** | NotificationService | Notify users and drivers of ride status |
| **Factory Pattern** | For creating Vehicle objects based on type (e.g., Sedan, SUV) |  |

**Step 5: SOLID Principles in Practice**

| **Principle** | **Application** |
| --- | --- |
| **S - Single Responsibility** | Each class handles one responsibility (User books, Payment processes) |
| **O - Open/Closed** | You can add new payment types without modifying PaymentService |
| **L - Liskov Substitution** | Driver and User can extend from Person base class if needed |
| **I - Interface Segregation** | Split interfaces if Admin, User, and Driver have different responsibilities |
| **D - Dependency Inversion** | High-level modules like RideService depend on abstractions (interfaces) |

**Interview Q&A Based on This LLD**

**Q1: How would you design the ride-matching algorithm?**

**Answer**:  
I'd implement a **RideMatchingService** class as a singleton that maintains a list of available drivers with location data. On a ride request, it finds the nearest driver using Euclidean distance or real-time traffic APIs (if scaling). It separates matching logic from User or Driver class (SRP).

**Q2: Which design patterns will you use and why?**

**Answer**:

* **Strategy Pattern** for payment types.
* **Singleton** for services like RideMatchingService and NotificationService.
* **Observer Pattern** to notify users and drivers on ride status change.

**Q3: How would you ensure the system is extensible?**

**Answer**:

* Use **interfaces/abstract classes** for services like payment so new payment types can be added without changing core logic.
* Follow **Open/Closed Principle** so modules are open for extension but closed for modification.
* Use **Factory pattern** to create vehicles without hardcoding types.

**Q4: How would you handle real-time driver updates?**

**Answer**:

* Maintain a **DriverLocationService** that receives location updates periodically (e.g., via WebSocket).
* Store active drivers in a geo-indexed data structure (e.g., QuadTree, Geohash) for fast lookup during matching.

**Q5: What would be your approach for unit testing this LLD?**

**Answer**:

* Use dependency injection to mock services like RideMatchingService and PaymentService.
* Test each class independently (SRP makes it easier).
* Validate matching logic, fare calculation, ride status transitions, and observer notifications.

**Bonus: Real-Time Extension Ideas**

* Add support for **carpooling** using Composite Pattern.
* Track driver ratings and block drivers below a threshold (extend Driver class).
* Add role-based access for Admin using Interface Segregation.

**Summary**

| **Concept** | **Application in LLD** |
| --- | --- |
| OOP | Classes for User, Driver, Ride |
| Design Patterns | Strategy, Singleton, Observer, Factory |
| SOLID | Followed in class separation and extensibility |
| UML Diagrams | Class diagram shows relationships |
| Interview Focus | Pattern usage, principles, extensibility, testing |

**Topic 2: History of Programming**

**1. Machine language**

**2. Assembly level language**

**3. Procedural Programming**

**4. Object-Oriented Programming**

**What is class and Objects**

**Topic 2: History of Programming**

**1. Machine Language**

**Definition:**

* Machine language is the **lowest-level programming language**, consisting of **binary code** (0s and 1s).
* It is **directly executed by the CPU**.
* Programs written in machine language are **hardware-specific** and **not human-readable**.

**Example:**

Binary Machine Code (for Intel CPU):

10110000 01100001 → MOV AL, 61h

This instruction moves the hexadecimal value 61 (which is ASCII for 'a') into register AL.

**👨‍💻 Real-World Use:**

Used in **embedded systems**, **bootloaders**, **hardware drivers**, and **firmware**.

**📌 Interview Question:**

**Q:** Why is machine language not used for general-purpose programming today?  
**A:** Because it's difficult to read, write, and debug, and is hardware-specific. Modern languages offer abstraction and portability.

**2. Assembly Level Language**

**Definition:**

* Assembly Language uses **mnemonics** and is a **low-level language** closer to machine code but **slightly human-readable**.
* Requires an **assembler** to convert into machine language.

**💡 Example:**

MOV AX, 5 ; Load 5 into AX register

ADD AX, 3 ; Add 3 to AX

**👨‍💻 Real-World Use:**

Used in **microcontrollers**, **performance-critical code**, and **reverse engineering**.

**📌 Interview Question:**

**Q:** How does Assembly differ from Machine Language?  
**A:** Assembly uses mnemonics and symbolic names (e.g., MOV, AX) for instructions instead of binary, making it more human-readable.

**3. Procedural Programming**

**Definition:**

* Procedural Programming is a **top-down** approach focused on **functions or procedures**.
* It executes instructions **sequentially**, often using **global data** and **function calls**.
* Popular languages: **C, Pascal, Fortran**

**💡 Simple Java Code in Procedural Style:**

*public class ProceduralExample {*

*public static void main(String[] args) {*

*int result = add(5, 3);*

*System.out.println("Sum is: " + result);*

*}*

*public static int add(int a, int b) {*

*return a + b;*

*}*

*}*

**👨‍💻 Real-World Use:**

Legacy systems, operating systems (e.g., Unix in C), system utilities.

**📌 Interview Questions:**

**Q:** What are the main features of procedural programming?  
**A:** Sequential execution, use of functions, global variables, and reusability of code through function calls.

**Q:** Is Java a procedural language?  
**A:** Java supports both procedural and object-oriented styles, but it's primarily object-oriented.

**4. Object-Oriented Programming (OOP)**

**Definition:**

* OOP is a **paradigm based on objects** which bundle **data and behavior**.
* Promotes **modularity, encapsulation, inheritance, and polymorphism**.
* Languages: **Java, Python, C++, C#**

**What is a Class?**

**Definition:**

* A **class** is a **blueprint** or **template** for creating objects.
* It defines **attributes (fields)** and **behaviors (methods)**.

**What is an Object?**

**Definition:**

* An **object** is an **instance** of a class that has **its own values** and can perform actions defined in the class.

**💡 Simple Java OOP Example:**

*// Class definition*

*class Car {*

*String brand;*

*int speed;*

*// Constructor*

*Car(String brand, int speed) {*

*this.brand = brand;*

*this.speed = speed;*

*}*

*// Method*

*void display() {*

*System.out.println("Brand: " + brand + ", Speed: " + speed);*

*}*

*}*

*// Main class*

*public class OOPExample {*

*public static void main(String[] args) {*

*Car car1 = new Car("Tesla", 120); // Object creation*

*car1.display();*

*}*

*}*

**📌 Interview Questions:**

**Q:** What are the 4 main pillars of OOP?  
**A:** Encapsulation, Inheritance, Polymorphism, and Abstraction.

**Q:** What is the difference between class and object?  
**A:** Class is a template; Object is an instance of the class.

**Q:** How does OOP promote reusability?  
**A:** Through inheritance, where a class can reuse fields and methods from another class.

**🧠 Summary Table**

| **Paradigm** | **Language Level** | **Code Example** | **Execution Style** | **Key Feature** |
| --- | --- | --- | --- | --- |
| Machine Language | Very Low | Binary | CPU-Direct | Fast, hardware-specific |
| Assembly Language | Low | Mnemonics | Assembled | Hardware control |
| Procedural Programming | Mid | Functions | Top-down | Reusability via functions |
| Object-Oriented (OOP) | High | Classes/Objs | Object-based | Modular, reusable, scalable |

**Bonus: Interview Practice Sheet**

| **Question** | **Answer** |
| --- | --- |
| What is a class in Java? | A class is a blueprint for creating objects with fields and methods. |
| What is procedural programming? | A method of writing code in sequential steps using functions. |
| Difference between OOP and procedural? | OOP uses objects and classes; procedural uses functions and global data. |
| Give a real-world example of object and class. | Class = Car design; Object = MyTeslaCar |
| Why is OOP preferred in large systems? | Due to its modularity, scalability, and ease of maintenance. |

**Topic 3: Pillars of OOPS**

1. **Abstraction**
2. **Encapsulation**
3. **Inheritance**
4. **Polymorphism**

Here’s a complete, **interview-ready explanation** of the **4 Pillars of Object-Oriented Programming (OOP)**, along with:

* Theory
* Real-world analogies
* Java code examples with comments
* Top interview questions with answers
* A **Low-Level Design (LLD) project** using a Car, ModernCar, and ElectricCar example

**🔰 Topic 3: Four Pillars of OOPS**

**🔹 1. Abstraction**

**Definition:**  
Abstraction means hiding internal implementation details and showing only the necessary features.

**Real-life Example:**  
When you drive a car, you use the steering wheel and pedals—you don’t need to know how the engine works internally.

**Java Code:**

**abstract class Car {**

**abstract void start(); // abstract method – no implementation**

**abstract void stop();**

**}**

**Interview Q&A:**

* **Q:** What is abstraction in Java?
* **A:** It's a way to expose essential features while hiding the internal details using abstract classes or interfaces.

**🔹 2. Encapsulation**

**Definition:**  
Encapsulation means binding data (variables) and methods into a single unit and restricting direct access to some of the object's components.

**Real-life Example:**  
A speedometer encapsulates the logic of showing speed. You can’t directly set the speed from outside.

**Java Code:**

**class Engine {**

**private int temperature; // private variable**

**public void setTemperature(int t) {**

**if (t > 0) temperature = t;**

**}**

**public int getTemperature() {**

**return temperature;**

**}**

**}**

**Interview Q&A:**

* **Q:** What is encapsulation in Java?
* **A:** Wrapping data and code together as a single unit and controlling access via access modifiers.

**🔹 3. Inheritance**

**Definition:**  
Inheritance allows one class to inherit properties and behaviors (methods) of another class.

**Real-life Example:**  
An ElectricCar is a type of Car and inherits the ability to start and stop.

**Java Code:**

**class Car {**

**void start() {**

**System.out.println("Car started");**

**}**

**}**

**class ElectricCar extends Car {**

**void chargeBattery() {**

**System.out.println("Battery charging...");**

**}**

**}**

**Interview Q&A:**

* **Q:** What is inheritance in Java?
* **A:** It enables a class to acquire properties and methods of another class, using extends keyword.

**🔹 4. Polymorphism**

**Definition:**  
Polymorphism allows us to perform a single action in different ways (many forms).

**Types:**

* **Compile-time (Method Overloading)**
* **Runtime (Method Overriding)**

**Polymorphism in Java**

Polymorphism means **“many forms”** — the ability of a single function, operator, or object to behave in different ways depending on the context.

There are **two types** of polymorphism in Java:

**🔹 1. Compile-time Polymorphism (Method Overloading)**

**What is it?**

Compile-time polymorphism occurs when **multiple methods have the same name but different parameters** (type, number, or sequence) within the same class.

**How it works:**

* The method to be executed is determined during **compilation**.
* Achieved using **method overloading**.

**Java Example:**

**class Calculator {**

**int add(int a, int b) {**

**return a + b;**

**}**

**double add(double a, double b) {**

**return a + b;**

**}**

**int add(int a, int b, int c) {**

**return a + b + c;**

**}**

**}**

**public class Main {**

**public static void main(String[] args) {**

**Calculator calc = new Calculator();**

**System.out.println(calc.add(2, 3)); // Output: 5**

**System.out.println(calc.add(2.5, 3.2)); // Output: 5.7**

**System.out.println(calc.add(1, 2, 3)); // Output: 6**

**}**

**}**

**Key Points:**

* Happens at **compile time**
* Same method name, **different signature**
* **No inheritance** required

**💬 Interview Questions**

**Q1:** What is method overloading in Java?  
**A:** Defining multiple methods with the same name but different parameter lists within the same class.

**Q2:** Can return type alone differentiate overloaded methods?  
**A:** ❌ No, the return type is not sufficient for overloading.

**🔹 2. Runtime Polymorphism (Method Overriding)**

**What is it?**

Runtime polymorphism occurs when a **subclass overrides a method of its parent class**. The method that gets called is determined **at runtime**, based on the object type.

**How it works:**

* Achieved using **inheritance** and **method overriding**
* The overridden method in the child class is called using a **parent class reference** pointing to the **child class object**

**Java Example:**

**class Vehicle {**

**void run() {**

**System.out.println("Vehicle is running");**

**}**

**}**

**class Car extends Vehicle {**

**@Override**

**void run() {**

**System.out.println("Car is running smoothly");**

**}**

**}**

**public class Main {**

**public static void main(String[] args) {**

**Vehicle obj = new Car(); // Upcasting**

**obj.run(); // Output: Car is running smoothly**

**}**

**}**

**📌 Key Points:**

* Happens at **runtime**
* Requires **inheritance**
* Uses **@Override** annotation
* Supports **dynamic method dispatch**

**💬 Interview Questions**

**Q1:** What is method overriding in Java?  
**A:** When a subclass provides a specific implementation of a method that is already defined in its parent class.

**Q2:** Can we override a static method?  
**A:** ❌ No. Static methods are **class-level** and can't be overridden, but they can be **hidden**.

**Q3:** What enables runtime polymorphism in Java?  
**A:** **Dynamic method dispatch** using method overriding and inheritance.

**Summary Table**

| **Feature** | **Method Overloading** | **Method Overriding** |
| --- | --- | --- |
| Type | Compile-time | Runtime |
| Requires Inheritance | ❌ No | ✅ Yes |
| Method Signature | Must be different | Must be same |
| Access Modifier | Can vary | Should not reduce visibility |
| Return Type | Can be different | Can be covariant |
| Static Methods | ✅ Can be overloaded | ❌ Cannot be overridden |

**Java Code:**

**class Car {**

**void drive() {**

**System.out.println("Driving a generic car");**

**}**

**}**

**class ElectricCar extends Car {**

**@Override**

**void drive() {**

**System.out.println("Driving an electric car silently");**

**}**

**}**

**Interview Q&A:**

* **Q:** What is polymorphism?
* **A:** It allows objects to behave differently based on their runtime type. Overloading is compile-time, overriding is runtime polymorphism.

**🚗 LLD Project: Car System with Modern & Electric Cars**

**📌 Goal:**

Design a modular Car system that uses all 4 OOP pillars.

**📦 Classes:**

1. Car (Abstract) - [Abstraction]
2. ModernCar extends Car - [Inheritance + Polymorphism]
3. ElectricCar extends ModernCar - [Inheritance + Method Overriding]
4. Engine - [Encapsulation]

**✅ Java Code:**

**// Abstraction using abstract class**

**abstract class Car {**

**protected String brand;**

**public Car(String brand) {**

**this.brand = brand;**

**}**

**abstract void start();**

**abstract void stop();**

**}**

**// Encapsulation: Engine class hides temperature logic**

**class Engine {**

**private int temperature;**

**public void setTemperature(int temp) {**

**if (temp >= 0 && temp <= 120) {**

**this.temperature = temp;**

**}**

**}**

**public int getTemperature() {**

**return this.temperature;**

**}**

**}**

**// Inheritance and Method Overriding**

**class ModernCar extends Car {**

**Engine engine = new Engine();**

**public ModernCar(String brand) {**

**super(brand);**

**}**

**@Override**

**void start() {**

**engine.setTemperature(70);**

**System.out.println(brand + " Modern Car started. Temp: " + engine.getTemperature());**

**}**

**@Override**

**void stop() {**

**System.out.println(brand + " Modern Car stopped.");**

**}**

**}**

**// Polymorphism via method overriding**

**class ElectricCar extends ModernCar {**

**public ElectricCar(String brand) {**

**super(brand);**

**}**

**@Override**

**void start() {**

**System.out.println(brand + " Electric Car started silently.");**

**}**

**public void chargeBattery() {**

**System.out.println(brand + " Battery charging...");**

**}**

**}**

**// Driver Class**

**public class Main {**

**public static void main(String[] args) {**

**Car tesla = new ElectricCar("Tesla"); // Polymorphic behavior**

**tesla.start(); // Calls overridden method**

**tesla.stop();**

**Car honda = new ModernCar("Honda");**

**honda.start();**

**honda.stop();**

**}**

**}**

**🧠 Summary Table (OOP Pillars)**

| **Pillar** | **Keyword/Usage** | **Example Class** | **Benefit** |
| --- | --- | --- | --- |
| Abstraction | abstract, interface | Car | Focus on what, not how |
| Encapsulation | private, public | Engine | Data protection |
| Inheritance | extends | ModernCar, ElectricCar | Code reuse |
| Polymorphism | Method Overriding/Overloading | start() method | Dynamic behavior |

**🎯 Top Interview Questions (with Answers)**

**Q1: How is abstraction implemented in Java?**

**A:** Through abstract classes and interfaces which hide implementation details and show only necessary operations.

**Q2: Give real-world use of encapsulation.**

**A:** Banking systems hide account balance logic and allow updates through secure methods.

**Q3: Difference between method overloading and overriding?**

**A:**

* **Overloading:** Same method name, different parameters (compile-time).
* **Overriding:** Subclass redefines superclass method (runtime).

**Q4: What are the advantages of OOP?**

**A:**

* Modularity
* Reusability
* Scalability
* Maintainability

**Topic 4: SOLID Design Principle**

**SOLID** principles, each with real-world examples and common interview questions & answers, plus a deep dive into **Liskov Substitution Principle (LSP)** rules like **signature rule, return type rule, exception rule, and method rule**.

**SOLID Principles — Detailed Explanation with Real-Time Examples & Interview Q&A**

**S — Single Responsibility Principle (SRP)**

**Definition:**  
A class should have **only one reason to change**, meaning it should have only one responsibility or job.

**Real-Time Example:**  
Imagine a class InvoiceProcessor that both calculates invoice totals and generates invoice PDFs. If tax laws change (affecting calculation), or the PDF layout changes, this class would have to change for two unrelated reasons. Better to separate calculation and PDF generation into two classes.

**Why it matters:**

* Easier to maintain and test
* Changes in one responsibility don’t affect others
* Improves code readability and reduces bugs

**Common Interview Q:**  
*Q:* Why should a class have only one responsibility?  
*A:* To keep code modular and maintainable, making changes easier and safer without affecting unrelated functionality.

**O — Open/Closed Principle (OCP)**

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

**Real-Time Example:**  
Consider a payment processing system that supports credit cards. Later, you want to add PayPal without modifying existing classes. You create a new PayPal payment class implementing a common Payment interface rather than changing existing code.

**Why it matters:**

* Encourages adding new features without breaking existing code
* Enhances system stability
* Supports scalability and growth

**Common Interview Q:**  
*Q:* How do you make your classes open for extension but closed for modification?  
*A:* By using abstractions like interfaces or abstract classes and extending them, instead of changing existing code.

**L — Liskov Substitution Principle (LSP)**

**Definition:**  
Objects of a superclass should be replaceable with objects of its subclasses **without affecting the correctness of the program**.

**Real-Time Example:**  
Suppose a base class Bird has a method fly(). A subclass Penguin cannot fly. If you substitute a Penguin where a Bird is expected, your program might break. This violates LSP. Instead, you could redesign: separate flying birds and non-flying birds into different hierarchies or interfaces.

**Important Rules of LSP**

LSP is the most subtle SOLID principle and has four key rules:

1. **Signature Rule:**  
   The method signatures in the subclass should match those in the base class exactly — same method name, same parameters.
2. **Return Type Rule:**  
   The return type in the subclass method must be compatible with the base class method return type (covariant return types allowed in some languages).
3. **Exception Rule:**  
   The subclass methods should not throw new or broader exceptions than the base class method. This ensures that clients can rely on the base class contract.
4. **Method Rule:**  
   Subclass methods must preserve the behavior expected by the base class — no breaking existing invariants or contracts.

**Why it matters:**

* Ensures substitutability of subclasses
* Prevents unexpected behavior and bugs in polymorphic use
* Maintains program correctness and robustness

**Common Interview Q:**  
*Q:* What happens if you violate the Liskov Substitution Principle?  
*A:* It can cause runtime errors or unexpected behavior when subclasses replace base classes, breaking polymorphism and leading to fragile code.

**I — Interface Segregation Principle (ISP)**

**Definition:**  
Clients should not be forced to depend on interfaces they do not use. Prefer many small, specific interfaces over a large, general-purpose interface.

**Real-Time Example:**  
Consider an all-encompassing Vehicle interface with methods like fly(), drive(), and sail(). A Car class that implements Vehicle must implement all methods, including fly() and sail(), which don’t apply. ISP encourages breaking this into separate interfaces: Drivable, Flyable, Sailable, so classes only implement what they need.

**Why it matters:**

* Avoids forcing unnecessary methods on clients
* Makes systems easier to maintain and extend
* Promotes cleaner, decoupled design

**Common Interview Q:**  
*Q:* Why is Interface Segregation important?  
*A:* Because it prevents bloated interfaces that force classes to implement irrelevant methods, reducing coupling and increasing flexibility.

**D — Dependency Inversion Principle (DIP)**

**Definition:**  
High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces or abstract classes). Also, abstractions should not depend on details; details should depend on abstractions.

**Real-Time Example:**  
A NotificationService sends messages via email or SMS. Instead of hardcoding email or SMS sending, NotificationService depends on an abstraction MessageSender. Concrete implementations like EmailSender or SmsSender are injected. This allows swapping message senders without modifying NotificationService.

**Why it matters:**

* Decouples high-level and low-level modules
* Makes code more flexible, testable, and maintainable
* Supports easy swapping of implementations (e.g., for testing or future changes)

**Common Interview Q:**  
*Q:* How do you implement Dependency Inversion in your code?  
*A:* By programming to interfaces and injecting dependencies (via constructor or setter), not instantiating concrete classes directly.

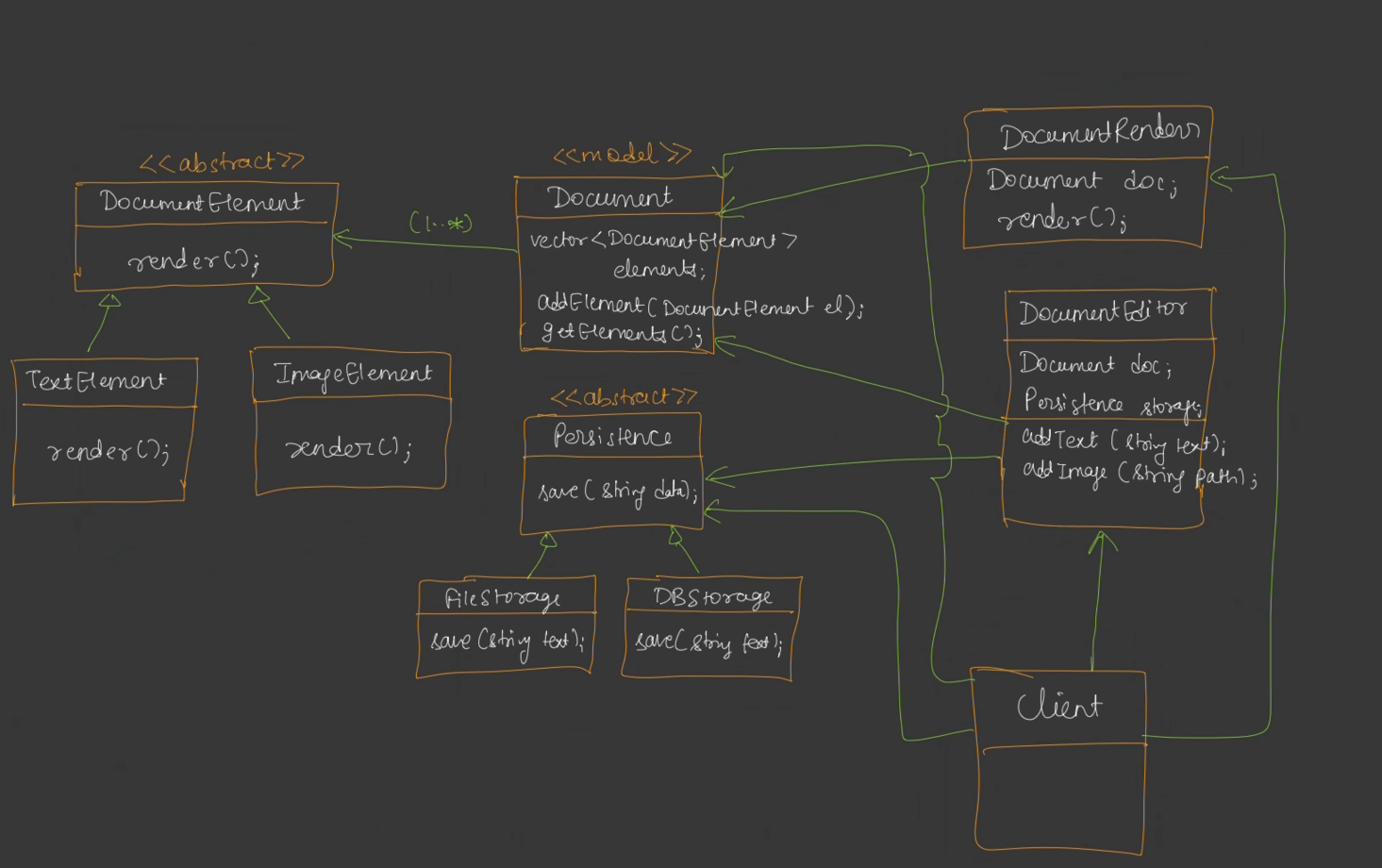
**Summary Table**

| **Principle** | **Focus** | **Real-World Example** | **Interview Q & A** |
| --- | --- | --- | --- |
| SRP | One class, one responsibility | Separate invoice calculation & PDF generation | Q: Why one responsibility? A: Easier maintenance |
| OCP | Open for extension, closed for modification | Add PayPal payment without changing existing code | Q: How to apply? A: Use abstractions, extend classes |
| LSP | Subclass must be substitutable for base | Penguin cannot fly if Bird has fly() method | Q: What if violated? A: Runtime errors, broken polymorphism |
| ISP | Many client-specific interfaces | Vehicle interfaces split into Drivable, Flyable | Q: Why? A: Avoid forcing unused methods |
| DIP | Depend on abstractions, not concrete classes | NotificationService depends on MessageSender interface | Q: How? A: Use interfaces + dependency injection |

**Extra Details on Liskov Substitution Principle (LSP) Rules**

* **Signature Rule:**  
  Subclass must implement the exact method signature to ensure polymorphism works seamlessly.
* **Return Type Rule:**  
  Return types should be compatible so that the calling code expects consistent output types.
* **Exception Rule:**  
  If base class method throws an exception, subclass methods can throw fewer or narrower exceptions but not new or broader exceptions, ensuring clients don’t face unexpected errors.
* **Method Rule:**  
  Subclass methods must honour the original method's behavior, preserving invariants, post-conditions, and side effects. For example, if a base method sorts a list, the subclass method shouldn't return an unsorted list.

**----1. Build Google Docs----**



Based on the provided Low-Level Design (LLD) diagram, here's how a "Build Google Docs" project would work:

This LLD outlines a modular and extensible architecture, primarily focusing on document rendering, editing, and persistence.

Core Components and Their Interactions:

* Document Element Hierarchy (Composite Pattern):
  + Document Element (Abstract): The base class for all individual elements within a document. It defines a common interface for rendering (render()).
  + Text Element: Concrete implementation for rendering text content.
  + Image Element: Concrete implementation for rendering image content.
  + Document (Model): This acts as a container for a collection of Document Elements (vector <DocumentElement> elements). It provides methods to add new elements (addElement(DocumentElement el)) and retrieve them (getElements()). This signifies the Composite design pattern, where a Document can be treated as a Document Element in terms of its ability to be rendered, even though it contains other elements. The (1..\*) multiplicity from Document-to-Document Element indicates that a Document contains one or more Document Elements.
* Document Rendering:
  + Document Render: This class is responsible for displaying the Document. It takes a Document object (Document doc) and has a render() method. This render() method would iterate through the Document Elements within the Document and call their respective render() methods, effectively rendering the entire document.
* Document Editing:
  + Document Editor: This is the central component for user interaction and document modification.
    - It holds a Document object (Document doc) which it manipulates.
    - It also has a Persistence object (Persistence storage), indicating its reliance on the persistence layer for saving and loading.
    - Key methods include addText(String text) and addImage(String path), allowing users to add content to the document. These methods would internally create Text Element and Image Element objects and add them to the Document's element collection.
* Persistence Layer (Strategy Pattern):
  + Persistence (Abstract): Defines an abstract interface for saving data (save(String data)). This allows for different storage mechanisms to be plugged in.
  + File Storage: Concrete implementation for saving the document content to a file. It has a save(String text) method.
  + DBS Storage: Concrete implementation for saving the document content to a database. It also has a save(String text) method.
  + The Document Editor interacts with the Persistence interface, decoupling it from the specific storage mechanism. This is an example of the Strategy design pattern, where the Document Editor can switch between different persistence strategies.
* Client:
  + The Client represents the application's entry point or the user interface. It interacts with the Document Editor to create, modify, and manage documents. It also interacts with the Document Render to display documents. The arrows from Client to Document Editor and Document Render indicate that the Client uses these components.

How the Project Works (Scenario):

1. Application Startup / User Interaction:
   * The Client (e.g., the Google Docs application interface) starts.
   * The Client can initiate the creation of a new document or open an existing one.
2. Creating a New Document:
   * The Client would create an instance of Document Editor.
   * The Document Editor would internally create a new, empty Document object.
3. Editing the Document:
   * The user interacts with the Client to add content (e.g., typing text, inserting images).
   * The Client calls methods on the Document Editor (e.g., editor.addText("Hello world!"), editor.addImage("/path/to/image.jpg")).
   * The Document Editor then:
     + Creates appropriate Text Element or Image Element objects.
     + Adds these new elements to its internal Document object's elements vector.
4. Rendering the Document (Displaying on Screen):
   * To display the current state of the document, the Client would create an instance of Document Render.
   * The Client passes the Document object (likely obtained from the Document Editor) to the Document Render.
   * The Document Render calls its render() method. This method iterates through all Document Elements within the Document and calls render() on each individual element (e.g., textElement.render(), imageElement.render()), effectively drawing the document content on the screen.
5. Saving the Document:
   * When the user saves the document, the Client would trigger a save operation on the Document Editor.
   * The Document Editor, having a Persistence object, would call its save() method.
   * Depending on the chosen persistence strategy (e.g., if File Storage was injected into the Document Editor), the File Storage object's save(String text) method would be invoked to write the document's serialized content to a file. If DBS Storage was used, it would save to the database. The Document Editor would need to serialize the Document object into a String format before passing it to the save method.
6. Loading an Existing Document:
   * (Not explicitly shown in methods but implied by persistence) To load, the Client would likely tell the Document Editor to load a document.
   * The Document Editor would use its Persistence object to retrieve the raw document data (as a String).
   * The Document Editor would then deserialize this String data back into a Document object, populating its elements vector with Text Element and Image Element instances as appropriate.

Key Design Principles Illustrated:

* Separation of Concerns: Different responsibilities (rendering, editing, persistence) are handled by distinct classes.
* Polymorphism: The render() method is defined in the abstract Document Element and implemented polymorphically by Text Element and Image Element. Similarly, save() is polymorphic for Persistence implementations.
* Abstraction: Document Element and Persistence are abstract classes, defining common interfaces without specifying implementation details.
* Composition: Document is composed of Document Elements.
* Dependency Injection (Implied): The Document Editor likely receives its Persistence object through some form of dependency injection, allowing for easy swapping of storage mechanisms.
* Strategy Pattern: The Persistence hierarchy allows for different storage strategies to be used interchangeably.
* Composite Pattern: The Document and Document Element hierarchy allows treating individual elements and collections of elements uniformly.

This LLD provides a solid foundation for a Google Docs-like application, offering flexibility for future extensions (e.g., adding more element types like tables, integrating with cloud services, version control).

1. **Strategy Design Pattern**

The Strategy Design Pattern is a behavioural design pattern that enables an algorithm's behavior to be selected at runtime. Instead of implementing a single algorithm directly, it allows a client to choose from a family of algorithms.

**What is the Strategy Design Pattern?**

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

**Key components of the Strategy Pattern:**

* **Context:** The class that maintains a reference to a Strategy object and uses it to perform some action. It does not know the concrete strategy being used, only the interface.
* **Strategy (Interface/Abstract Class):** Declares an interface common to all supported algorithms. The Context uses this interface to call the algorithm defined by a Concrete Strategy.
* **Concrete Strategy:** Implements the Strategy interface, providing a specific implementation of the algorithm.

**Why use the Strategy Design Pattern?**

1. **Eliminates Conditional Logic (if-else or switch-case statements):** Without the Strategy pattern, you might end up with large conditional blocks in your client code to decide which algorithm to execute. The Strategy pattern replaces these conditionals with a single method call to the current strategy object.
2. **Improves Maintainability:** Each algorithm is encapsulated in its own class, making it easier to modify, test, and debug individual algorithms without affecting others.
3. **Enhances Flexibility and Extensibility:** New algorithms can be added easily by creating new concrete strategy classes without modifying existing client code or other strategy classes.
4. **Promotes Reusability:** Algorithms can be reused across different contexts as long as they conform to the Strategy interface.
5. **Separation of Concerns:** It separates the algorithm's implementation from the client that uses it, adhering to the Single Responsibility Principle.

**Where to use the Strategy Design Pattern?**

* When you have many related classes that differ only in their behavior.
* When you need different variants of an algorithm, and you want to switch between them at runtime.
* When you want to avoid coupling the client directly to the implementation of the algorithms.
* When an algorithm uses data that clients shouldn't know about, or when an algorithm has multiple variants that are difficult to manage with conditional statements.

**How to use the Strategy Design Pattern?**

1. **Identify the varying behavior:** Determine which parts of your system represent different algorithms or behavior that might change.
2. **Define the Strategy Interface:** Create an interface or abstract class that declares a common method(s) for all concrete strategy implementations.
3. **Implement Concrete Strategies:** Create separate classes for each algorithm, implementing the Strategy interface.
4. **Create the Context Class:** Design a context class that holds a reference to a Strategy object. It should have a method that delegates the work to the current strategy.
5. **Allow Strategy Injection:** The context class should allow the client to set or change the strategy at runtime, typically through its constructor or a setter method.

**Problems with Inheritance and Solutions**

While inheritance (specifically implementation inheritance) is a fundamental concept in Object-Oriented Programming, it comes with certain problems, particularly when used for behavior variation:

1. **Tight Coupling:** Subclasses are tightly coupled to their parent classes. Changes in the superclass can force changes in all subclasses, leading to fragility.
2. **Limited Flexibility:** Once a class inherits from another, its behavior is fixed at compile-time. It's difficult to change or extend its behavior at runtime.
3. **"Fragile Base Class" Problem:** Modifications to a base class can inadvertently break the functionality of its subclasses, even if the subclasses themselves haven't been changed.
4. **Behavioural Bloat (Inheritance Hierarchy Explosion):** If you try to achieve behavioural variation solely through inheritance, you might end up with a deep and wide inheritance hierarchy (many subclasses) just to account for different combinations of behaviors. This becomes unmanageable.
5. **Inability to Change Behavior at Runtime:** The inherited behavior is fixed when the object is instantiated. You cannot change an object's behavior dynamically.
6. **Violation of Single Responsibility Principle:** A base class might end up being responsible for too many things if it tries to define all possible behaviors or common behaviors for all its subclasses.

**Solutions provided by the Strategy Design Pattern (and composition in general):**

The Strategy Pattern offers a better solution to behavioral variation by favouring **composition over inheritance**:

* **Loose Coupling:** Instead of inheriting behavior, the Context class *contains* a reference to a Strategy object. This means the Context is not coupled to a specific concrete strategy, only to the Strategy interface.
* **Runtime Flexibility:** The Context can change its behavior by swapping out the Strategy object at runtime. This is impossible with traditional inheritance where behavior is fixed.
* **Encapsulation of Algorithms:** Each algorithm is encapsulated in its own class, making it easier to manage and modify.
* **Open/Closed Principal Adherence:** New algorithms can be added without modifying existing code (Context or other strategies). This means the system is "open for extension, but closed for modification."
* **Avoids Inheritance Hierarchy Explosion:** Instead of creating numerous subclasses to represent different behaviors, you create a flat hierarchy of strategy classes.

1. **Factory Design Pattern**

**What is Factory Design Pattern?**

The **Factory Design Pattern** is a **creational design pattern** that provides an **interface or method for creating objects**, allowing subclasses or a factory class to decide **which class to instantiate**. The main idea is to **delegate object creation logic** to a separate method or class, **instead of using direct object instantiation (new) in client code**.

**Why Factory Design Pattern?**

The Factory Design Pattern is useful because:

* 🔸 **Encapsulates object creation**: Hides the instantiation logic from the client.
* 🔸 **Promotes loose coupling**: The client code depends on an interface, not on specific classes.
* 🔸 **Improves scalability**: You can introduce new types without changing the client logic.
* 🔸 **Reduces code duplication**: Centralizes creation logic in one place.
* 🔸 **Supports Open/Closed Principle**: Easy to extend with new types without modifying existing code.

**When to Use Factory Design Pattern?**

You should consider using a factory design pattern when:

* 🔹 You have **a lot of classes implementing the same interface or extending a common superclass**, and you need to decide which class to instantiate at runtime.
* 🔹 The **creation logic is complex or repetitive**.
* 🔹 You want to **centralize or standardize object creation**.
* 🔹 You want your system to be **easily extendable** for future object types.

**Types of Factory Design Patterns**

**🔹 1. Simple Factory Pattern (also known as Static Factory)**

* Not part of official GoF (Gang of Four) design patterns.
* Uses a **single static method** in a factory class to create instances based on input.
* Often implemented with if-else or switch statements.
* Useful for **basic object creation** when there are only a few types.

**Use when**:

* You want a simple centralized place to create objects.
* You don’t want the client to use new directly.

**🔹 2. Factory Method Pattern (GoF pattern)**

* Defines a **method in a superclass or interface** for creating an object, but allows **subclasses to alter the type of objects that will be created**.
* Promotes the **Open/Closed Principle**.
* Typically used with **polymorphism**: the client uses the superclass/interface, and the actual object is created by the subclass.

**Use when**:

* You want to **delegate object creation to subclasses**.
* You need to add **new types frequently without modifying existing code**.
* Your object creation requires **specific logic depending on the subclass**.

**🔹 3. Abstract Factory Pattern (GoF pattern)**

* Used to create **families of related or dependent objects** (e.g., GUI elements for different OS themes).
* Provides an **interface for creating a set of objects**, each factory implementation creates a different variant of the family.
* Ensures that the related objects are **used together**.

**Use when**:

* Your system needs to be **platform-independent or theme-based**.
* You want to create **groups of related objects** with consistent behavior.
* You want to **ensure compatibility** among the created objects.

**Real-World Applications**

| **Application Area** | **Use of Factory Pattern** |
| --- | --- |
| Spring Framework | Bean creation using Factory Beans |
| JDBC API | Uses factory method DriverManager.getConnection() |
| GUI Frameworks | Creating platform-dependent UI components |
| Logging Frameworks | Abstracting creation of different logger types |
| Android SDK | View inflation, service creation |
| Game Development | Creating characters, levels, or weapons at runtime |
| AI & ML Pipelines | Factory methods to switch between models dynamically |

**Summary Table**

| **Pattern Type** | **Description** | **Example Use Case** |
| --- | --- | --- |
| Simple Factory | Static method to create objects | Creating different shapes based on input |
| Factory Method | Subclass decides which object to create | Notification system (Email, SMS, Push) |
| Abstract Factory | Create families of related objects | UI components for different OS themes |

1. **Singleton Design Pattern**

**same comprehensive explanation of the Singleton Design Pattern**, now **enhanced with real-world applications and examples** relevant to **Low-Level Design (LLD)** and **System Design**.

**✅ 1. What is Singleton Design Pattern?**

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

It restricts instantiation of a class to one object and is typically used for shared resources such as configuration, logging, or caching in an application.

**✅ 2. Why use Singleton Design Pattern?**

**🔹 Key Reasons:**

* You want to avoid multiple instances of a **resource-heavy class**.
* You need a **central manager or coordinator** class.
* You want to maintain a **consistent state** across modules/services.
* You want to enforce **controlled access** to a shared resource.

**✅ 3. Where to use Singleton Design Pattern? (Real-World Applications)**

| **Application** | **Real-World Usage** | **Why Singleton?** |
| --- | --- | --- |
| **Logger System** | All modules use the same logger instance | To avoid duplicate file handlers or log files |
| **Database Connection Manager** | JDBC/ORM/NoSQL connection pools | To maintain a single pool across the system |
| **Configuration Manager** | Load and access configuration across modules | To ensure consistent configs (e.g., API keys, DB URL) |
| **Cache Manager** | In-memory caches like Redis, Memcached wrappers | Central access to the same cache |
| **Thread Pool Manager** | Java ExecutorService, custom thread pools | Prevent unbounded thread creation |
| **Service Registry (in Microservices)** | Tracks services like Eureka/Consul | One global service discovery instance |
| **Authentication Manager** | Auth tokens/session management | One global session/token tracker |
| **Analytics Tracker** | Unified analytics for logging usage, metrics | Same instance across all requests for uniformity |
| **Print Spooler (OS Level)** | Managing print jobs | One queue manager per system |
| **Operating System Kernel** | Handles interrupts, resource management | Single access point for hardware-software interaction |
| **WebDriver in Selenium (Test Framework)** | Only one browser driver instance per test suite | Avoid multiple browsers opening at once |

**✅ 4. How Singleton Design Pattern Works?**

To implement Singleton:

1. Make the **constructor private**.
2. Store the **instance in a private static variable**.
3. Provide a **public static method (e.g., getInstance())** to access the instance.

This pattern ensures:

* **Only one instance** is ever created.
* It is **shared** wherever required.
* Object creation is **deferred** (lazy) or **eager**, based on need.

**✅ 5. When to Use Singleton Design Pattern?**

Use Singleton when:

* You need **exactly one object** for coordination or control.
* You want to **reduce memory overhead** by sharing the same object.
* You want to **centralize access** to a shared resource or service.
* You’re building a **stateless service** or managing **global state**.

**✅ 6. Advantages in System Design**

| **Advantage** | **Description** |
| --- | --- |
| ✅ **Centralized Control** | One shared object handles coordination (like DB manager) |
| ✅ **Memory Efficient** | Prevents repeated object creation |
| ✅ **Thread-safe Access** | When implemented correctly, supports concurrent access |
| ✅ **Global State Maintenance** | Shared configuration, session, or cache |
| ✅ **Encapsulation** | Controls object creation from one place |

**✅ 7. Drawbacks / Cautions in Large Systems**

| **Issue** | **Explanation** |
| --- | --- |
| ❌ **Global State** | Hard to debug if state changes unexpectedly |
| ❌ **Tight Coupling** | All modules become dependent on the Singleton |
| ❌ **Difficult to Unit Test** | Hard to mock or isolate |
| ❌ **Thread Safety Risks** | Bad implementation can break in multi-threaded systems |
| ❌ **Violation of SOLID Principles** | May break Single Responsibility or Dependency Inversion principles if overused |

**✅ 8. Best Practices in LLD / System Design**

* 🔹 Make Singleton **thread-safe** (especially for shared resources).
* 🔹 Use **lazy initialization** if creation is expensive.
* 🔹 Combine with **Dependency Injection** for testability.
* 🔹 Avoid making Singleton classes **do too many things** (violate SRP).
* 🔹 Use Singleton only if the instance truly must be shared across the system.

**✅ 9. Interview-Oriented Summary**

| **Topic** | **Description** |
| --- | --- |
| Pattern Type | Creational |
| Purpose | Only one instance per class |
| Access | Through static method (e.g., getInstance()) |
| LLD Use Cases | Logger, Cache, DB Pool, Config Manager |
| Benefits | Resource-efficient, globally accessible |
| Limitations | Global state, testing difficulty, tight coupling |
| Thread-Safe? | Yes, if implemented properly |
| Testability | Needs careful mocking or interface-based Singleton |

**✅ 10. Analogy (Real-Life Example)**

**President of a Country**:

* Only **one active president** at a time.
* All citizens refer to the same person.
* There’s a **central authority** and **shared decision-maker**.

**Like a Singleton**, there is only **one centralized access point** for leadership.