**Steps to follow for a LLD question**

1. Clarify requirements and core use cases.
2. Identify the main entities/objects we need. This can be done by writing down the requirements in sentences and whatever nouns are encountered we can treat it as a entity/class. Also identify their attributes which each of this class would have. In case of multiple classes see how they would relate with each other.
3. Check for the methods which need to be implemented
4. Exception Handling where check for unexpected inputs, errors and edge cases

### **Best practices**

### **Favor composition over inheritance**

* **Use Interfaces and Delegation**  
  Instead of creating a subclass, define behavior via interfaces and inject the behavior via composition.

**✅ Example:**

interface Engine {

void start();

}

class DieselEngine implements Engine {

public void start() {

System.out.println("Diesel engine starting...");

}

}

class Car {

private final Engine engine;

public Car(Engine engine) {

this.engine = engine;

}

public void startCar() {

engine.start();

}

}

**Usage:**

Engine diesel = new DieselEngine();

Car car = new Car(diesel);

car.startCar();

➡ This way, you can change the engine without touching the Car class. That’s composition.

**❌ What Not to Do (Inheritance approach):**

java

CopyEdit

class Engine {

void start() {

System.out.println("Engine starting...");

}

}

class Car extends Engine {

void startCar() {

start();

}

}

* **Favor Behavior Injection Over Extension**

You can create different behaviors as pluggable components.

#### Example:

interface FlyBehavior {

void fly();

}

class CanFly implements FlyBehavior {

public void fly() {

System.out.println("Flying high!");

}

}

class CannotFly implements FlyBehavior {

public void fly() {

System.out.println("Can't fly.");

}

}

class Bird {

private FlyBehavior flyBehavior;

public Bird(FlyBehavior flyBehavior) {

this.flyBehavior = flyBehavior;

}

public void performFly() {

flyBehavior.fly();

}

public void setFlyBehavior(FlyBehavior fb) {

this.flyBehavior = fb;

}

}

**Usage:**  
  
public class Main {

public static void main(String[] args) {

// A bird that can fly

FlyBehavior canFly = new CanFly();

Bird sparrow = new Bird(canFly);

System.out.print("Sparrow: ");

sparrow.performFly(); // Output: Flying high!

// A bird that can't fly

FlyBehavior cannotFly = new CannotFly();

Bird penguin = new Bird(cannotFly);

System.out.print("Penguin: ");

penguin.performFly(); // Output: Can't fly.

// Dynamically change behavior at runtime

System.out.print("Penguin suddenly learns to fly: ");

penguin.setFlyBehavior(new CanFly());

penguin.performFly(); // Output: Flying high!

}

### } What This Shows:

* Bird class doesn't care **how** flying is implemented.
* You can **swap behavior at runtime** (penguin.setFlyBehavior(new CanFly())).
* You avoid the rigid structure of inheritance (Bird doesn’t need to extend a FlyingBird or NonFlyingBird).