**Day 21 – July 26th**

**TASK 1 – LISKOV SUBSTITUTION PRINCIPLE**

**Violation of VIKSOV**

abstract class Bird {

    abstract void fly ();

}

class Eagle extends Bird {

    @Override

    public void fly () {

        System.out.println(" Eagles fly");

    }

}

class Ostrich extends Bird {

    @Override

    public void fly () { // dummy implentation

        System.out.println("can’t fly high but It lays big egg");

    }

}

class Driverclass{

    public staic void main(String[] args){

    }

}

**TASK 2 – Implementation of VISKOV**

abstract class BirdsthatFly {

    abstract void fly() ;

}

abstract class BirdsthatDontFly {

    abstract void Speciality() ;

}

class Eagle extends BirdsthatFly {

    @Override

    public void fly() {

        System.out.println(" Eagles fly");

    }

}

class Ostrich  extends BirdsthatDontFly {

    @Override

    public void Speciality() {

        System.out.println ("It lays big egg");

    }

}

class Driverclass{

    public static void main(String[] args){

    }

}

**INTERFACE SEGREGATION PRINCIPLE**

**TASK 3 – Violation of Interface Segregation Principle**

interface ICalcShapesArea {

    calcArea();

    calcVolume();

}

class Circle implements ICalcShapesArea {

    calcArea() { sout()};

    calcVolume() { sout()}; // dummy implentation as it been forced

}

class Sphere implements ICalcShapesArea {

    calcArea() { sout()};

    calcVolume() { sout()};

}

class Driverclass {

    public static void main (  ,...  ) {

}

**TASK 4 - Implementation of Interface Segregation Principle**

interface ICalcArea {

    calcArea(); calcPerimeter();

}

interface ICalcVolume {

    calcVolume();

}

class Circle implements ICalcArea {

    @Override

    calcArea() { sout()};

}

class Sphere implements ICalcArea, ICalcVolume {

     @Override

     calcArea() { sout()};

     calcVolume() { sout()};

}

class Driverclass {

    psvm(  ,...  ) {

    }

}

**HOMETASK1: Can you create a code based on the below diagrammatic representation?**

import java.util.List;  
  
// General Interface (segregated based on usage)  
interface Payment {  
 Object status();  
 List<Object> getListOfPayment();  
}  
  
// Bank-specific interface  
interface Bank {  
 void initiatePayment();  
 void stopPayment();  
}  
  
// Loan-specific interface  
interface Loan {  
 void initiateRepayment();  
 void initiateFinalSettlement();  
}  
  
// Class that uses Bank operations + Payment details  
class BankPayment implements Payment, Bank {  
 public Object status() {  
 System.*out*.println("Bank Payment Status: SUCCESS");  
 return "BankPaymentStatus";  
 }  
  
 public List<Object> getListOfPayment() {  
 System.*out*.println("Fetching bank payments...");  
 return List.*of*("BankTxn1", "BankTxn2");  
 }  
  
 public void initiatePayment() {  
 System.*out*.println("Initiating bank payment...");  
 }  
  
 public void stopPayment() {  
 System.*out*.println("Stopping bank payment...");  
 }  
}  
  
// Class that uses Loan operations + Payment details  
class LoanPayment implements Payment, Loan {  
 public Object status() {  
 System.*out*.println("Loan Payment Status: PENDING");  
 return "LoanPaymentStatus";  
 }  
  
 public List<Object> getListOfPayment() {  
 System.*out*.println("Fetching loan payments...");  
 return List.*of*("LoanTxnA", "LoanTxnB");  
 }  
  
 public void initiateRepayment() {  
 System.*out*.println("Initiating repayment...");  
 }  
  
 public void initiateFinalSettlement() {  
 System.*out*.println("Final settlement initiated...");  
 }  
  
 // Specific loan-related method (not in interface)  
 public List<Object> initiateLoanSettlement() {  
 System.*out*.println("Initiating loan settlement...");  
 return List.*of*("SettlementTxnX");  
 }  
}  
public class Day21\_InterfaceSeg1 {  
 public static void main(String[] args) {  
 // Test BankPayment  
 System.*out*.println("---- Bank Payment ----");  
 BankPayment bankPayment = new BankPayment();  
 bankPayment.initiatePayment();  
 bankPayment.stopPayment();  
 System.*out*.println("Status: " + bankPayment.status());  
 System.*out*.println("Payments: " + bankPayment.getListOfPayment());  
  
 // Test LoanPayment  
 System.*out*.println("\n---- Loan Payment ----");  
 LoanPayment loanPayment = new LoanPayment();  
 loanPayment.initiateRepayment();  
 loanPayment.initiateFinalSettlement();  
 System.*out*.println("Status: " + loanPayment.status());  
 System.*out*.println("Payments: " + loanPayment.getListOfPayment());  
 System.*out*.println("Loan Settlements: " + loanPayment.initiateLoanSettlement());  
 }  
}

**Output**

---- Bank Payment ----

Initiating bank payment...

Stopping bank payment...

Bank Payment Status: SUCCESS

Status: BankPaymentStatus

Fetching bank payments...

Payments: [BankTxn1, BankTxn2]

---- Loan Payment ----

Initiating repayment...

Final settlement initiated...

Loan Payment Status: PENDING

Status: LoanPaymentStatus

Fetching loan payments...

Payments: [LoanTxnA, LoanTxnB]

Initiating loan settlement...

Loan Settlements: [SettlementTxnX]

**HOMETASK2**

class Animal {

void sound() {

sout(" sounds of different animals");

}

}

class Cat extends Animal{

@Override

void sound() {

sout(" Meow is the sound of cat");

}

}

class Main{

psvm(String[] args) {

Animal obj = new Cat();

obj.sound(); //Meow is the sound of cat

}

}

issue with Substitution  and Generics

Java Generics -- it has introduced  a challenge - substitution principle...

 is cat a subtype of Animal,  List<cat> is not a subtype of List<Animal>

List<Cat> Cobj = new ArrayList<>();

List<Animal> Aobj = Cobj;  ===// this will give you a wildcard ,

import java.util.\*;  
  
class Animal {  
 void sound() {  
 System.*out*.println("Generic animal sound");  
 }  
}  
  
class Cat extends Animal {  
 @Override  
 void sound() {  
 System.*out*.println("Meow is the sound of cat");  
 }  
}  
  
class Dog extends Animal {  
 @Override  
 void sound() {  
 System.*out*.println("Woof is the sound of dog");  
 }  
}  
  
public class Day21\_GenericWildCards {  
 public static void main(String[] args) {  
 List<Cat> cats = new ArrayList<>();  
 cats.add(new Cat());  
  
 // Read-only wildcards  
 List<? extends Animal> animals = cats;  
 for (Animal a : animals) {  
 a.sound(); // ✅ Allowed  
 }  
  
 // animals.add(new Dog()); // ❌ Not allowed  
  
 // Write-safe wildcards  
 List<? super Cat> writeList = new ArrayList<>();  
 writeList.add(new Cat()); // ✅ OK  
 writeList.add(new Tiger()); // ✅ if Tiger extends Cat  
  
 // Object obj = writeList.get(0); // Only returns Object  
 }  
}  
  
class Tiger extends Cat {}

**Notes :**

* List<Cat>: A list that stores only Cat objects.
* new ArrayList<>(): Creates an empty list.
* add(new Cat()): Adds one Cat object to the list.
* This list can be any type that **extends Animal** — like Cat, Dog, or Tiger."
* It **accepts List<Cat>**, but **prevents adding** anything (read-only).
* Why? Because Java doesn’t know the **exact** subtype — it could be List<Dog>, and adding a Cat would be unsafe.
* Java **blocks adding** to a List<? extends Animal> because it doesn’t know what exact type the list holds.
* If it's List<Cat>, adding a Dog would break type safety.

**Substitution and Generic issues:**

* Cat extends Animal **JAVA doesn’t allow** List<Cat> is NOT a subtype of List<Animal>.

**Example that fails:**

List<Cat> catList = new ArrayList<>();

List<Animal> animalList = catList; //Compile-time error

Because **Java Generics are invariant**:

List<Cat> is not a subtype of List<Animal> — even though Cat is a subtype of Animal.

**Example if it allows:**

List<Cat> catList = new ArrayList<>();

List<Animal> animalList = catList; // Hypothetically allowed

animalList.add(new Dog()); // You just added a Dog to a List<Cat>!

This breaks **type safety** — so Java forbids this to protect your program.

**Solution for this is to Use WILDCARDS**

**READ-ONLY**

List<? extends Animal> animals = new ArrayList<Cat>();

animals.get(0); // Safe to read

// animals.add(new Dog()); Not allowed to add

You can **read**, but **cannot add** because Java doesn’t know the exact subtype.

**WRITE-SAFE**

List<? super Cat> animals = new ArrayList<Animal>();

animals.add(new Cat()); // Safe to write

animals.add(new Tiger()); // if Tiger extends Cat

Object obj = animals.get(0); // Only returns Object

You can **add**, but **can't safely read as Cat** (only Object)

**Unbounded WildCards**

**What is <?>?**

It’s a **wildcard with no bounds** — it means:

“It don’t care what type this list holds.”

You use <?> when:

* You **don't need to know** or use the **actual generic type**
* You're just **reading** values as Objects

List<?> — you can read (as Object), **cannot write**.

Useful for **generic printing**, logging, debugging.

Think of <?> as: *“any list of something, but I don’t care what*

**TASK3 – UnBounded WildCards**

void printList(List<?>  list)

{

for(Object element: list)

{

sout (element);

}

}

List<Cat> clist = new ArrayList<>();

clist.add(new Cat()); // Allowed: List<Cat> matches List<?>

printList(clist); //

* Accepts any type of list: List<Cat>, List<Dog>, List<String>, etc.
* Iterates through it and prints each element.
* Each item is treated as an Object, because the exact type is unknown.

✅ Safe, because:

* we’re not modifying (adding to) the list.
* we’re just printing.
* list.add(new Cat()); // NOT ALLOWED

We can't add, because:

* Java doesn't know what exact type? is.
* It might be List<Dog>, and you're trying to add a Cat.

Only null is allowed to be added to <?> lists.

**TASK 4 – UpperBounded WildCards**

void animalSound(List<? extends Animal> animalList) {

for(Animal elements : animalList

elements.sound();

}

}

List<Cat> cats = new ArrayList<>();

cats.add(new Cat());

animalSound(cats); //meow

**Modified Code:**

import java.util.\*;

class Animal {

void sound() {

System.out.println("Generic animal sound");

}

}

class Cat extends Animal {

@Override

void sound() {

System.out.println("Meow");

}

}

public class Main {

public static void main(String[] args) {

List<Cat> cats = new ArrayList<>();

cats.add(new Cat());

animalSound(cats); // ✅ Meow

}

static void animalSound(List<? extends Animal> animalList) {

for (Animal element : animalList) {

element.sound(); // Polymorphism in action

}

}

}

**Notes:**

**void animalSound(List<? extends Animal> animalList) {**

**for (Animal element : animalList) {**

**element.sound(); // ✅ Polymorphic call**

**}**

**}**

* animalList could be a List<Cat>, List<Dog>, etc.
* The sound() method works polymorphically.

**List<Cat> cats = new ArrayList<>();**

**cats.add(new Cat());**

**animalSound(cats); // Meow**

* Cat extends Animal, so List<Cat> is a valid argument for List<? extends Animal>.
* This allows the function to **read** safely from any subtype list of Animal.

**animalList.add(new Dog()); // Compile error!**

* Even though Dog extends Animal, Java forbids this because:

It doesn't know the actual list type.

If the actual list is List<Cat>, adding a Dog would be unsafe.

Upper-bounded wildcards (<? extends T>) are used when:

* We **only need to read** from the list.
* We’re dealing with a list of **T or its subtypes**.
* We want **polymorphic behavior**.

**TASK 5 – Lower Bounded Wildcards**

lower Bounded Wildcards

void addAcat(List<? super Cat> cats) {

cats.add(new Cat());

}

List<Animal> animals = new ArrayList<>();

addAcat(animals); //

**Modified code**

import java.util.\*;

class Animal {

void sound() {

System.out.println("Generic animal sound");

}

}

class Cat extends Animal {

void sound() {

System.out.println("Meow");

}

}

public class Main {

public static void main(String[] args) {

List<Animal> animals = new ArrayList<>();

addAcat(animals); // ✅ Adds a Cat to the list

animals.get(0).sound(); // ✅ Prints "Meow"

}

static void addAcat(List<? super Cat> cats) {

cats.add(new Cat()); // ✅ Safe to add Cat

}

}

**Notes:**

List<? super Cat> means:

* “A list of Cat or any of its **superclasses** (like Animal, Object)”
* This ensures it's safe to **add a Cat or a subclass of Cat** into the list.

**void addAcat(List<? super Cat> cats) {**

**cats.add(new Cat());**

**}**

* This method accepts a List that can store **Cat or its superclasses**.
* Since you're adding a Cat, it is **always safe**:
* If it's a List<Animal> → valid (Cat is an Animal)
* If it's a List<Object> → valid (Animal → Object)

**List<Animal> animals = new ArrayList<>();**

* create a list of Animal objects.
* This matches the requirement of List<? super Cat>, since Animal is a **superclass** of Cat.

**addAcat(animals); // Works!**

* passing a List<Animal> to a method that accepts List<? super Cat>.
* This is **legal and type-safe**.
* Internally, it adds a Cat to the animals list.

**Object obj = cats.get(0); // OK**

**Cat c = cats.get(0); // Compile error**

* The compiler **doesn’t know** what the actual list holds.
* It might be a List<Object> or List<Animal>, so it can only **guarantee Object type** on reading.

**TASK 6 – Tight Coupling**

class Student {

    public int roll\_no = 10;

    //private int roll\_no = 10;

    public int getRoll() {

        System.out.println("getRoll method");

        return roll\_no;

    }

    public void setRoll(int roll) {

        if(!(roll > 100))

            roll\_no = roll;

    }

}

class Tight\_Coupling01 {

    public static void main(String[] args) {

        Student sobj = new Student();

        sobj.roll\_no = 10;

        //sobj.roll\_no = 110;

        System.out.println("the roll no of student is "+ sobj.roll\_no); // 110

    }

}

**Notes**

**Tight Coupling** happens when **one class is highly dependent on another class’s implementation**. This means changes in one class may directly affect the other, making code harder to maintain, test, and reuse.

**Notes:**

**There is public roll\_no as 10. When the roll\_no is called in main class as sobj.roll\_no=110, it prints 110 as output, not as 10. And we also set the rule as ( !roll\_no>100).**

**sobj.roll\_no = 110;**

* Even though you have validation in setRoll(), this line **bypasses it** by directly modifying roll\_no.
* That’s the **core problem with tight coupling**:
* The main class is **tightly coupled** to the internal data of Student.
* **Encapsulation is violated**, making it hard to control changes and maintain integrity.

**TASK 7 – Loose Coupling**

class Student {  
 private int roll\_no = 0;  
  
 public int getRoll() {  
 System.*out*.println("getRoll method");  
 return roll\_no;  
 }  
  
 public void setRoll(int roll) {  
 if (!(roll > 100)) { // ✅ Missing closing parenthesis fixed  
 roll\_no = roll;  
 }  
 }  
}  
class Day21\_LooseCoupling {  
 public static void main(String[] args) {  
 Student sobj = new Student(); // ✅ Object created  
  
 sobj.setRoll(10); // ✅ Valid value passed  
  
 System.*out*.println("The roll no of student is " + sobj.getRoll()); // ✅ Print correctly  
 }  
}

**Output**

getRoll method

The roll no of student is 10

**Notes:**

**Loose Coupling** means that **classes are independent** and **minimally dependent** on each other. A loosely coupled system is one where components (classes/modules) can interact **without knowing much about each other's internal details**.

* Student class has **private data** (good encapsulation).
* main() method only interacts with it via **getter/setter**.
* If the internal structure of Student changes later, you **don’t need to change** Loose\_Coupling class — that's **loose coupling**.

**TASK 8 – Violation Of Dependency Inversion Principle**

class LightBulb {

void turnOn() {

System.out.println("Light turned on");

}

void turnOff() {

System.out.println("Light is off");

}

}

class Switch { // ❌ Directly depends on LightBulb

LightBulb lbulbobj;

Switch(LightBulb lbulbobj) {

this.lbulbobj = lbulbobj;

}

void operate() {

lbulbobj.turnOn();

}

public static void main(String[] args) {

LightBulb lbulbobj = new LightBulb(); // ✅ Fixed: class name corrected

Switch switchObj = new Switch(lbulbobj);

switchObj.operate();

}

}

**Notes**

* Switch class **depends on a concrete class (LightBulb)**.
* If you later want to turn on a **Fan, Heater, or SmartLight**, you'd need to modify the Switch class — this **breaks the Open/Closed Principle** too.
* It’s **tightly coupled** to LightBulb.

**TASK 9 – Implementation Of DIP**

interface SwitchOnOff {  
 void turnOn();  
 void turnOff();  
}  
  
// LightBulb now implements the SwitchOnOff interface  
class LightBulb implements SwitchOnOff {  
 public void turnOn() {  
 System.*out*.println("Light turned on");  
 }  
  
 public void turnOff() {  
 System.*out*.println("Light is off");  
 }  
}  
  
// The Switch depends on the abstraction, not on a concrete class  
class Switch {  
 private SwitchOnOff device;  
  
 // ✅ Proper constructor (not a method)  
 public Switch(SwitchOnOff device) {  
 this.device = device;  
 }  
  
 public void operate() {  
 device.turnOn();  
 }  
}  
public class Day21\_DIP\_Implementation {  
 public static void main(String[] args) {  
 SwitchOnOff lbulbobj = new LightBulb(); // ✅ Abstract reference  
 Switch lightswitch = new Switch(lbulbobj); // ✅ DIP: depends on abstraction  
 lightswitch.operate(); // ✅ Output: Light turned on  
 }  
}

**Output**

Light turned on

**Notes:**

* Switch depends only on the SwitchOnOff **interface**, not on LightBulb.
* You can now plug in a Fan, WashingMachine, SmartTV, etc. — **without modifying** Switch class.
* This satisfies **Open/Closed Principle** too: open for extension, closed for modification.

**TASK 10 – Dependency**

class Engine {

func start() {

print("Engine starting")

}

}

class Car {

func drive() {

let engine = Engine() // 🔴 Dependency

engine.start()

print("Car is driving")

}

}

// Usage

let myCar = Car()

myCar.drive()

**Notes:**

A **dependency** is any object that another object needs in order to function.  
In this case:

* The Car class **depends** on the Engine class because:
  + It **creates** an Engine object.
  + It **calls** the start() method of that object inside the drive() method.

let engine = Engine()

* The Car class is **hardcoded** to use the Engine class.
* If you ever want to:
  + Replace Engine with a HybridEngine or ElectricEngine
  + Test Car without starting a real engine

You’d have to **change the Car class** itself. That's **tight coupling**.

**TASK 11 – Aggregation**

**Aggregation** is a type of **association** that represents a "**has-a**" relationship, but with **weaker coupling**:

* **"Car has a Driver"**
* The **lifetime of the Driver is independent** of the Car.
* If the Car is destroyed, the Driver **still exists**.
* class Driver {
* var name: String
* init(name: String) {
* self.name = name
* }
* }
* class Car {
* var driver: Driver? // Aggregation
* init(driver: Driver?) {
* self.driver = driver
* }
* }
* // Usage
* let driver = Driver(name: "John")
* let myCar = Car(driver: driver)

**Notes:**

**class Driver {**

**var name: String**

**init(name: String) {**

**self.name = name**

**}**

**}**

* Driver is a separate class representing a person who drives.
* Has its own identity (name) and lifecycle.

**class Car {**

**var driver: Driver? // ✅ Aggregation — Car \*has\* a Driver, but does not own it**

**init(driver: Driver?) {**

**self.driver = driver**

**}**

**}**

* Car has an optional reference to a Driver.
* Car **does not create** the Driver itself — it just receives one.

**// Usage**

**let driver = Driver(name: "John") // ✅ Create a driver (independent)**

**let myCar = Car(driver: driver) // ✅ Assign driver to car (aggregation)**

* The Driver is created **outside** the Car.
* If the Car is deallocated, the Driver **still exists** in memory if referenced elsewhere

**TASK 12 – Composition**

**Composition** is a strong form of association where:

* One class **owns** objects of another class.
* If the **owning object is destroyed**, the owned objects are also destroyed.
* Think of it as a **“part-of”** relationship.
* class Wheel {
* init() {
* print("Wheel created")
* }
* }
* class Car {
* var wheels: [Wheel] // Composition
* init() {
* self.wheels = [Wheel(), Wheel(), Wheel(), Wheel()] // Wheels are created and owned by Car
* }
* }
* // Usage
* let myCar = Car() // All wheels created with the car

**Notes:**

**class Wheel {**

**init() {**

**print("Wheel created")**

**}**

**}**

* Wheel is a simple class representing a part of a car.

**class Car {**

**var wheels: [Wheel] // ✅ Composition relationship**

**init() {**

**self.wheels = [Wheel(), Wheel(), Wheel(), Wheel()] // Car creates the wheels**

**}**

**}**

* Car **creates** and **owns** 4 Wheel objects inside its constructor.
* Car is **fully responsible** for the lifecycle of its Wheels.
* If Car gets deallocated, its wheels will also be deallocated.

**let myCar = Car() // 🚗 All wheels are created immediately with the car**

* Car owns the Wheel objects completely.
* Wheel has **no existence outside** the Car.