Day 21 26th July 2025

SOLID:

S -

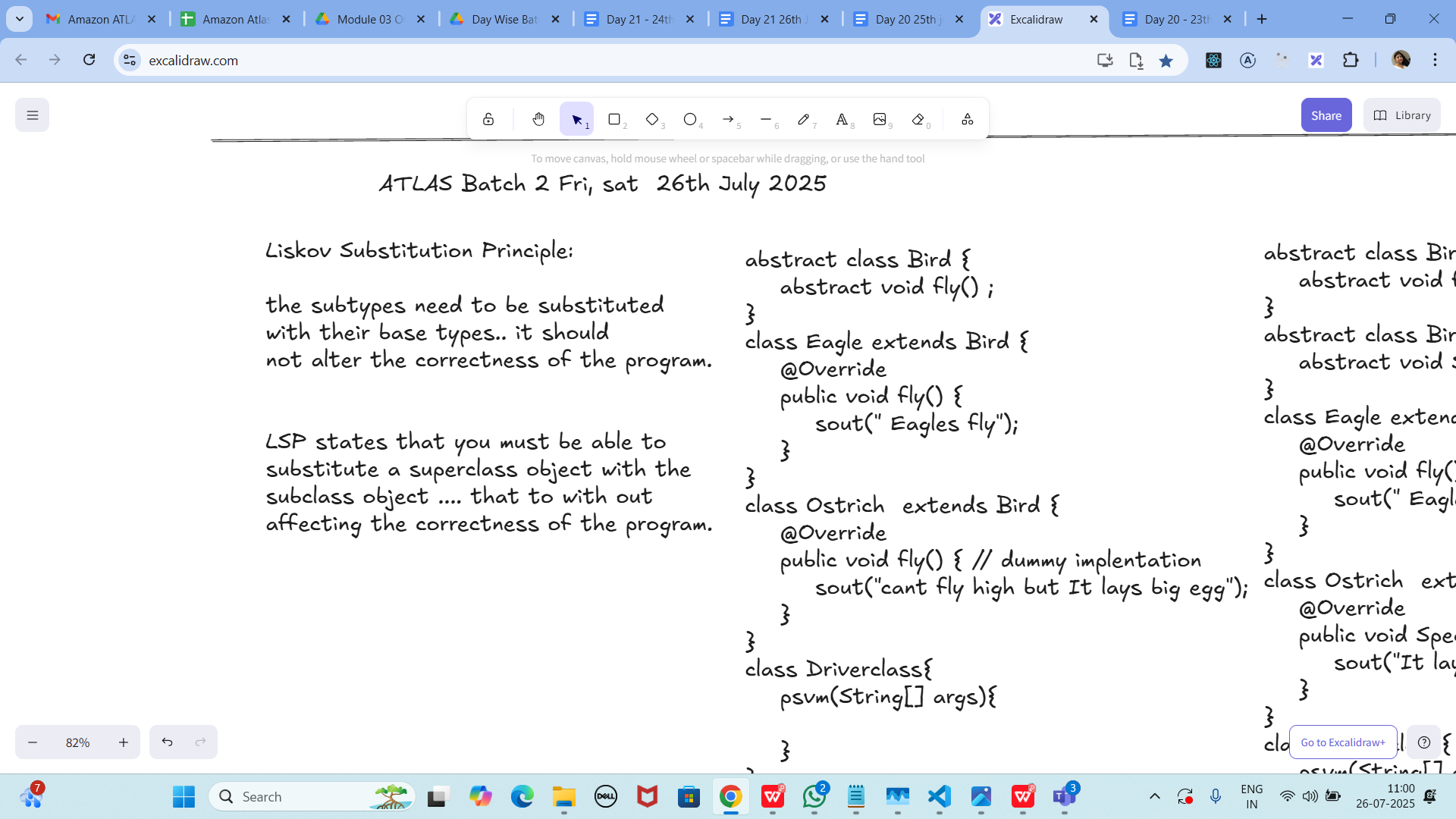
O -

L -

I -

D -

Liskov Substitution Principle



Subtypes must be usable anywhere their base type is expected without changing

program correctness or expected behavior (no weakened preconditions, no strengthened

postconditions).

Task 01:

Violation of Liskov

abstract class Bird {

abstract void fly() ;

}

class Eagle extends Bird {

@Override

public void fly() {

sout(" Eagles fly");

}

}

class Ostrich extends Bird {

@Override

public void fly() { // dummy implentation

sout("cant fly high but It lays big egg");

}

}

class Driverclass{

psvm(String[] args){

}

}

**Answer**: Violates LSP: Ostrich cannot fulfill fly(). Fix by separating behaviors (e.g., interface

Flyable { void fly(); }) or distinct hierarchies for flying/non-flying birds.

Task 02:

Implementation of Liskov

abstract class BirdsthatFly {

abstract void fly() ;

}

abstract class BirdsthatDontFly {

abstract void Speciality() ;

}

class Eagle extends BirdsthatFly {

@Override

public void fly() {

sout(" Eagles fly");

}

}

class Ostrich extends BirdsthatDontFly {

@Override

public void Speciality() {

sout("It lays big egg");

}

}

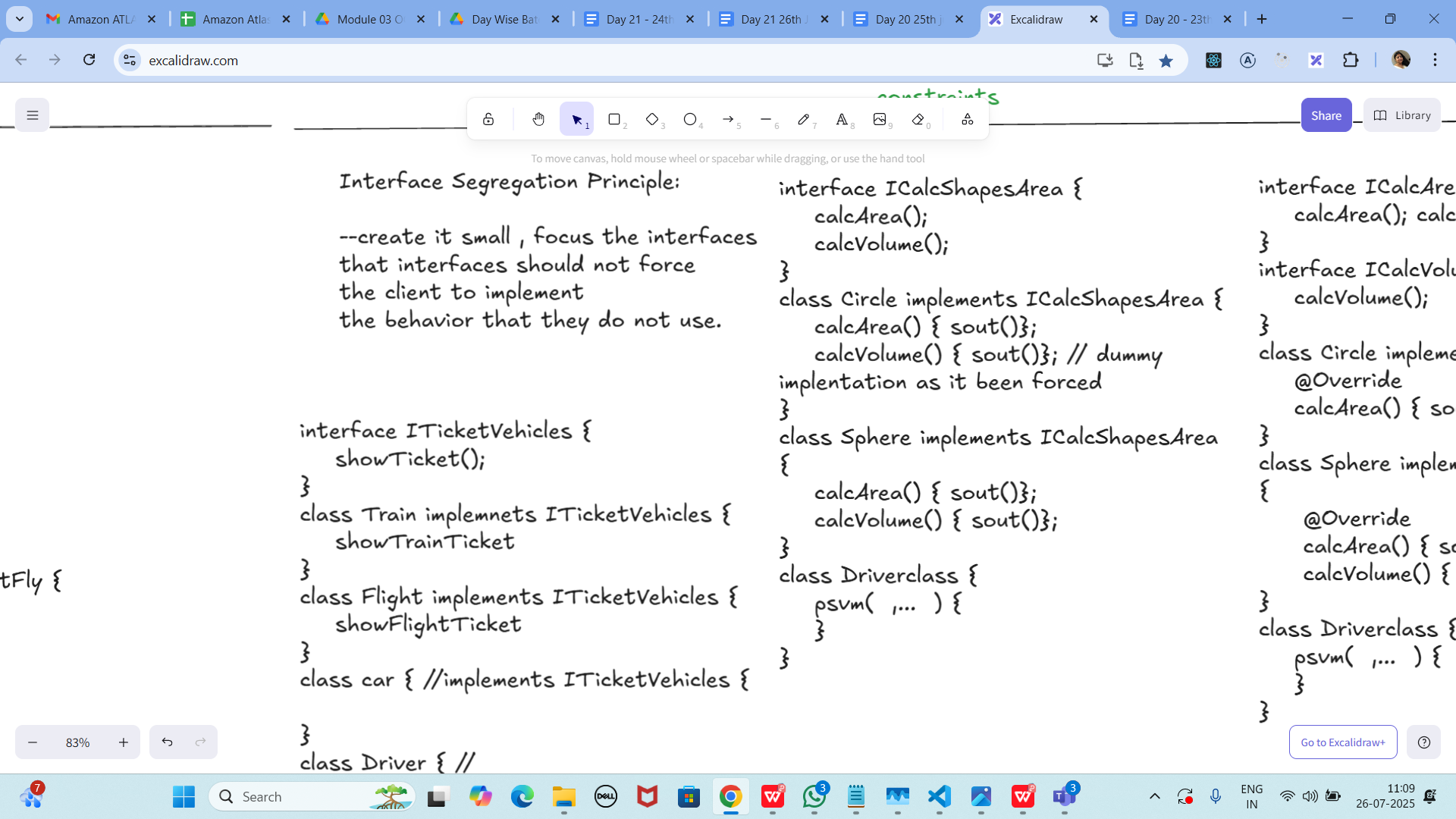
class Driverclass{

psvm(String[] args){

}

}

I - Interface Segregation Principle



**Answer** (correct form):

interface Flyable { void fly(); }

abstract class Bird { /\* common bird props \*/ }

class Eagle extends Bird implements Flyable {

public void fly(){ System.out.println(&quot;Eagles fly&quot;); }

}

class Ostrich extends Bird {

public void speciality(){ System.out.println(&quot;Lays big eggs&quot;); }

}

I - Interface Segregation Principle

Task 03:

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 {

psvm( ,... ) {

}

}

**Answer**: Circle is forced to implement calcVolume(). Split interfaces so 2D shapes don’t

implement 3D behavior.

Task 04:

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( ,... ) {

}

}

Answer (correct signatures):

interface ICalcArea { double calcArea(); }

interface ICalcPerimeter { double calcPerimeter(); }

interface ICalcVolume { double calcVolume(); }

class Circle implements ICalcArea, ICalcPerimeter {

public double calcArea(){ return Math.PI\*r\*r; }

public double calcPerimeter(){ return 2\*Math.PI\*r; }

private final double r; Circle(double r){ this.r=r; }

}

class Sphere implements ICalcArea, ICalcVolume {

public double calcArea(){ return 4\*Math.PI\*r\*r; }

public double calcVolume(){ return 4.0/3.0\*Math.PI\*r\*r\*r; }

private final double r; Sphere(double r){ this.r=r; }

}

Dependency Inversion Principle:

Answer: High-level modules should depend on abstractions, not concrete implementations;

abstractions shouldn’t depend on details—details depend on abstractions.

Dip violation code:

DIP - Dependency inversion Principle - Violation:

public class Clothes {

void seeRating() {

}

void viewSample() {

}

}

public class Cupboard { // high level class

// completly depending on low level class

//called clothes, books, vessels etc..

Clothes cobj;

void addClothes(Clothes cobj) {

}

void CustomizeClothes() {

}

}

customer asks to add books in the code:

public class Books {

void seeRating() {

}

void readSample() {

}

}

Answer: High-level Cupboard couples to concrete classes (Clothes, Books). Introduce a common

abstraction and depend on it.

Now DIP implementation:

Implementing Dependency Inversion Principle

public interface IProduct {

void SeeReviews();

void getSample();

}

public class Clothes implements IProduct {

@Override

public void SeeReviews() {

}

@Override

public void getSample() {

}

}

public class Books implements IProduct {

@Override

public void SeeReviews() {

}

@Override

public void getSample() {

}

}

Answer (add high-level using abstraction):

class Cupboard {

private final List&lt;IProduct&gt; items = new ArrayList&lt;&gt;();

void add(IProduct p){ items.add(p); }

void demoAll(){ items.forEach(IProduct::getSample); }

}

Relationship between classes:

Answer:

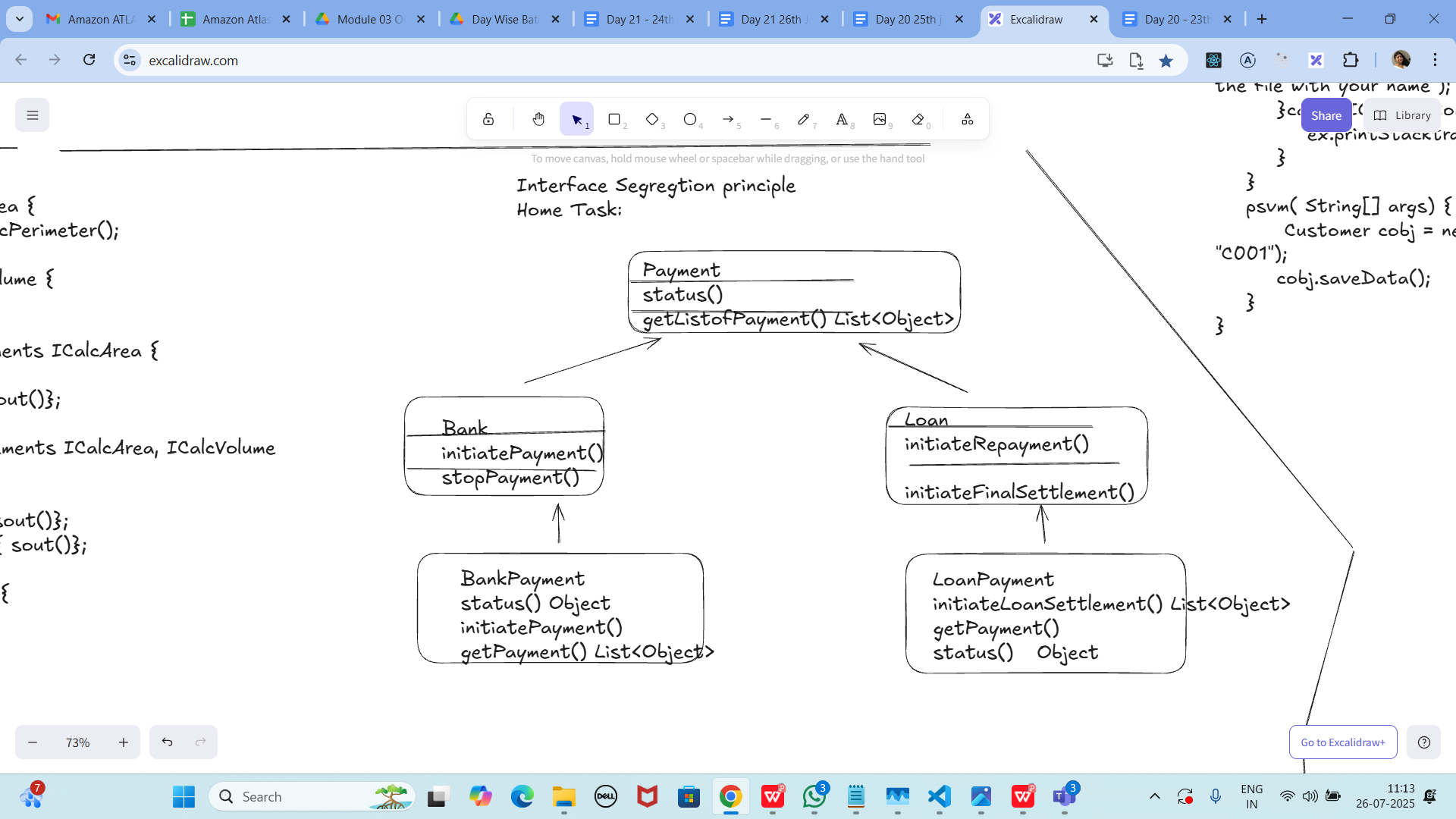
 Dependency: uses another class temporarily.

 Aggregation: whole–part, weak ownership (part can outlive whole).

 Composition: whole–part, strong ownership (part’s lifecycle tied to whole).

================================================================================================================================================

Home Tasks:



Home Tasks:

Task 01:

Can you create a code based on the below diagrammatic representation

Task 02:

class Animal {

void sound() {

sout(&quot; sounds of different animals&quot;);

}

}

class Cat extends Animal{

@Override

void sound() {

sout(&quot; Meow is the sound of cat&quot;);

}

}

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&lt;cat&gt; is not a subtype of List&lt;Animal&gt;

List&lt;Cat&gt; Cobj = new ArrayList&lt;&gt;();

List&lt;Animal&gt; Aobj = Cobj; ===// this will give you a wildcard ,

wildcards:

1. unbounded Wildcard

? ===&gt; any data type if you want to use you can use ?

2. Bounded Wildcards with an upper bound (? extends Type)

===&gt; yopu will use it when you need to accept a type and its own subtypes

3. Bounded Wildcards with an lower bound (? super Type)

===&gt; you will use when ypou need accept a type and its super types

Unbounded wildcards:

Task 03:

they are useful when the code does not depends on the actual type parmeter

void printList(List&lt;?&gt; list) {

for(Object element: list) {

sout (element);

}

}

List&lt;Cat&gt; clist = new ArrayList&lt;&gt;();

clist.add(new Cat());

printList(clist); //

Answer: Correct—use List&lt;?&gt; when logic is type-agnostic.

Task 04:

Upper Bounded Wildcards

void animalSound(List&lt;? extends Animal&gt; animalList) {

for(Animal elements : animalList

elements.sound();

}

}

List&lt;Cat&gt; cats = new ArrayList&lt;&gt;();

cats.add(new Cat());

animalSound(cats); //meow

Answer (fixed):

void animalSound(List&lt;? extends Animal&gt; animalList){

for(Animal a : animalList){ a.sound(); }

}

Task 05:

lower Bounded Wildcards

void addAcat(List&lt;? super Cat&gt; cats) {

cats.add(new Cat());

}

List&lt;Animal&gt; animals = new ArrayList&lt;&gt;();

addAcat(animals); //

Answer: Correct—? super Cat allows adding Cat into List&lt;Cat|Animal|Object&gt;.

—----------------------------

Task 06:

class Student {

public int roll\_no = 10;

//private int roll\_no = 10;

public int getRoll() {

System.out.println(&quot;getRoll method&quot;);

return roll\_no;

}

public void setRoll(int roll) {

if(!(roll &gt; 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(&quot;the roll no of student is &quot;+ sobj.roll\_no); // 110

}

}

Answer: Tight coupling via public field—encapsulate with private field + getters/setters only.

Task 07:

// loose coupling

class Student {

private int roll\_no = 0;

public int getRoll() {

sout(&quot;getRoll method&quot;);

return roll\_no;

}

public void setRoll(int roll) {

if(!(roll &gt; 100)

roll\_no = roll;

}

}

class Loose\_coupling {

psvm(String....) {

Student sobj = new Student();// Person pobj = new Student(); // person got a bonus

sobj.setRoll(10);

sout(&quot;the roll no of student is &quot;+ sobj.getroll();

}

}

Answer (fixed):

class Student {

private int roll\_no;

public int getRoll(){ System.out.println(&quot;getRoll method&quot;); return roll\_no; }

public void setRoll(int roll){ if(roll&lt;=100) roll\_no=roll; }

}

class Loose\_coupling {

public static void main(String... args){

Student s = new Student();

s.setRoll(10);

System.out.println(&quot;the roll no of student is &quot;+ s.getRoll());

}

}

Task 08:

DIP - violation:

class LightBulb {

void turnOn() {

sout(&quot;light turned on&quot;);

}

void turnOff() {

sout(&quot;light is off&quot;);

}

}

class Switch { // switch class directly depends on the lightbulb class ---- DIP violating

LightBulb lbulbobj;

Switch(LightBulb lbulbobj) {

this.lbulbobj = lbulbobj;

}

void operates(){

lbulbobj.turnOn();

}

psvm(....){

LightBulb lbulbobj = new lightBulb();

Switch Switchobj = new Switch(lbulbobj);

Switchobj.operate();

}

}

Answer: Violates DIP—Switch depends on concrete LightBulb.

Task 09:

DIP implementation:

interface SwitchOnOff {

void turnOn();

void turnOff(); // void remoteControl();// void alexaVoiceControl();

}

class LightBulb {

void turnOn() {

sout(&quot;light turned on&quot;);

}

void turnOff() {

sout(&quot;light is off&quot;);

}

} // or class fan, class inverter, class washing machine...(in future remote for washing machine

// i can extend without modification..

class Switch { // switch is depending on switchonoff class not on light bulb..

SwitchOnOff device;

void Switch(SwitchOnOff device) {

this.device = device;

}

void operates() {

device.turnOn();

}

}

class DIP {

psvm(....){

SwitchOnOff lbulbobj = new LightBulb();

Switch lightswitch = new Switch(lbulbobj);

lightswitch.operate();

}

}

Answer (fixed):

interface SwitchOnOff { void turnOn(); void turnOff(); }

class LightBulb implements SwitchOnOff {

public void turnOn(){ System.out.println(&quot;light turned on&quot;); }

public void turnOff(){ System.out.println(&quot;light is off&quot;); }

}

class Switch {

private final SwitchOnOff device;

Switch(SwitchOnOff device){ this.device=device; }

void operate(){ device.turnOn(); }

}

Task 10

Refer the below doc in Docs to study

Doc 07 UML Class Diagrams

Dependency

Answer: A uses B (method params/local vars). Weak, transient link.

Task 11:

Aggregation

Answer: Whole–part (shared ownership). Parts can exist independently (e.g., Team—Player).

Task 12:

Composition

Answer: Whole–part (strong ownership). Parts’ lifecycle bound to whole (e.g., House—Room).

================================================================================================================================================

==============================================================

**Info box:**

=============================================================

<https://excalidraw.com/#json=FvHREzZ_RxqrbO0_4iF3U,0WLjacoEfdym6RTsA9sGYA>

Updated link: @ 11.00 26th July 2025 is the below..

<https://excalidraw.com/#json=etbysGBZ-YBvF_bavIIil,E7NdpUtrmOPY-fn81ADwrg>

==============================================================

=============================================================

Info Box