Day 21 26th July 2025

SOLID:

S -

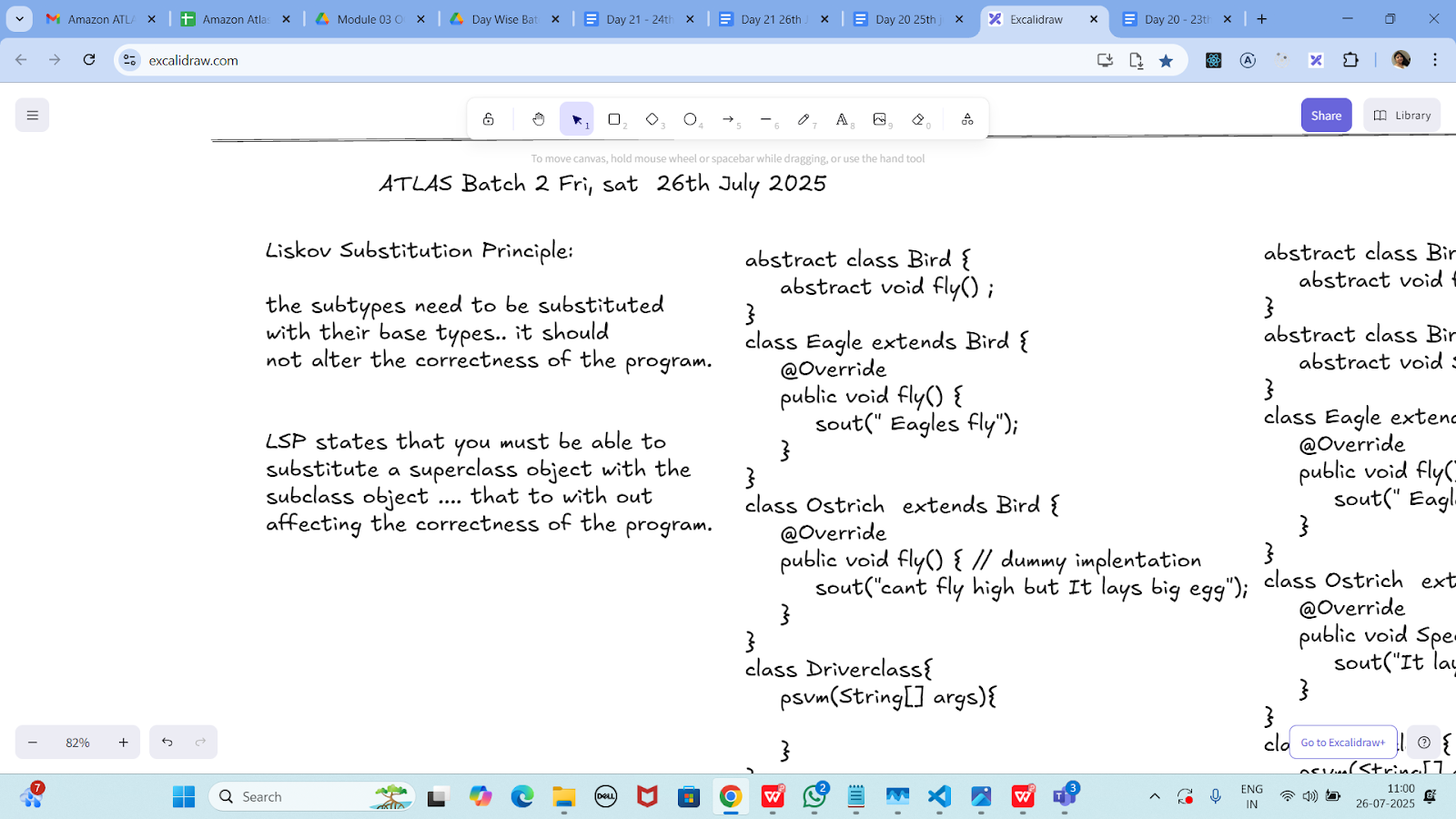
O -

L -

I -

D -

Liskov Substitution Principle



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){

}

}

abstract class Bird {

abstract void fly();

}

class Eagle extends Bird {

@Override

public void fly() {

System.*out*.println("Eagle flies high in the sky.");

}

}

class Ostrich extends Bird {

@Override

public void fly() {

// Violation: Ostrich can't fly

System.*out*.println("Ostrich can't really fly, but this method is here just to satisfy the abstract method.");

}

}

public class BirdLSPViolation {

public static void main(String[] args) {

Bird b1 = new Eagle();

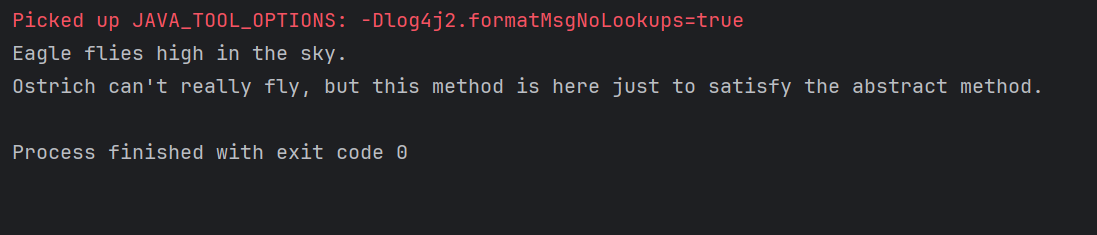
Bird b2 = new Ostrich();

b1.fly(); // OK

b2.fly(); // Logically incorrect - misleading behavior

}

}



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){

}

}

abstract class Bird {

abstract void fly();

}

class Eagle2 extends Bird {

@Override

public void fly() {

System.*out*.println("Eagle2 flies high in the sky.");

}

}

class Ostrich2 extends Bird {

@Override

public void fly() {

// Violation: Ostrich2 can't fly

System.*out*.println("Ostrich2 can't really fly, but this method is here just to satisfy the abstract method.");

}

}

public class BirdLSPViolation {

public static void main(String[] args) {

Bird b1 = new Eagle2();

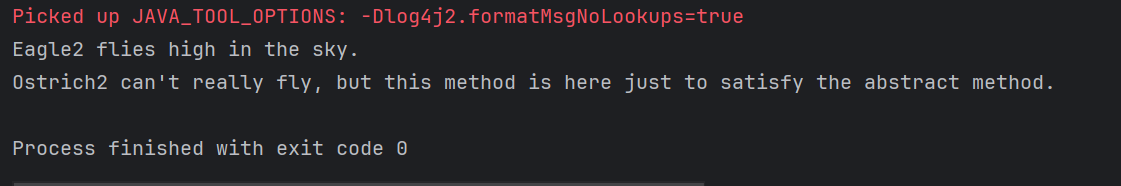
Bird b2 = new Ostrich2();

b1.fly(); // OK

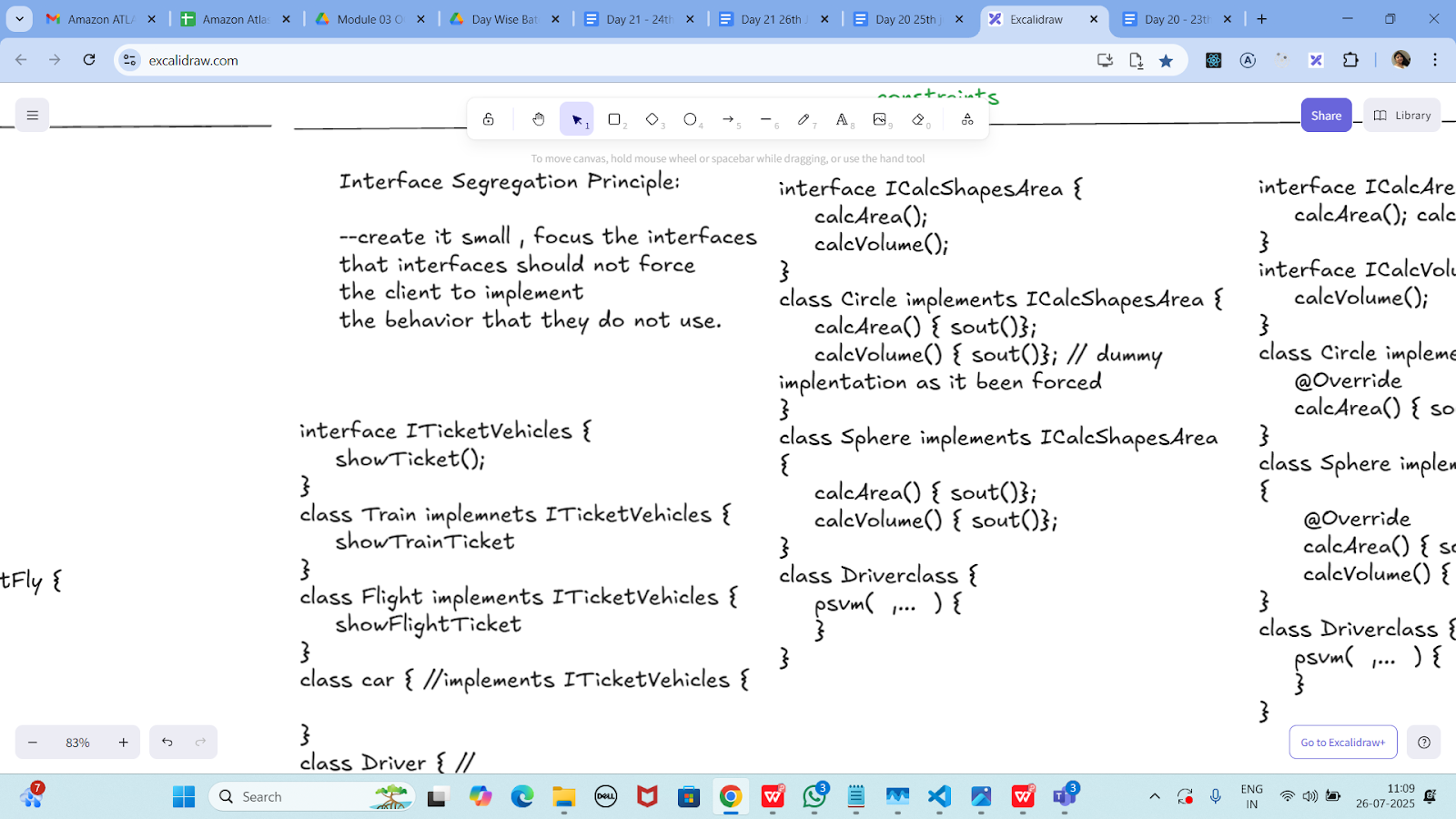
b2.fly(); // Logically incorrect - misleading behavior

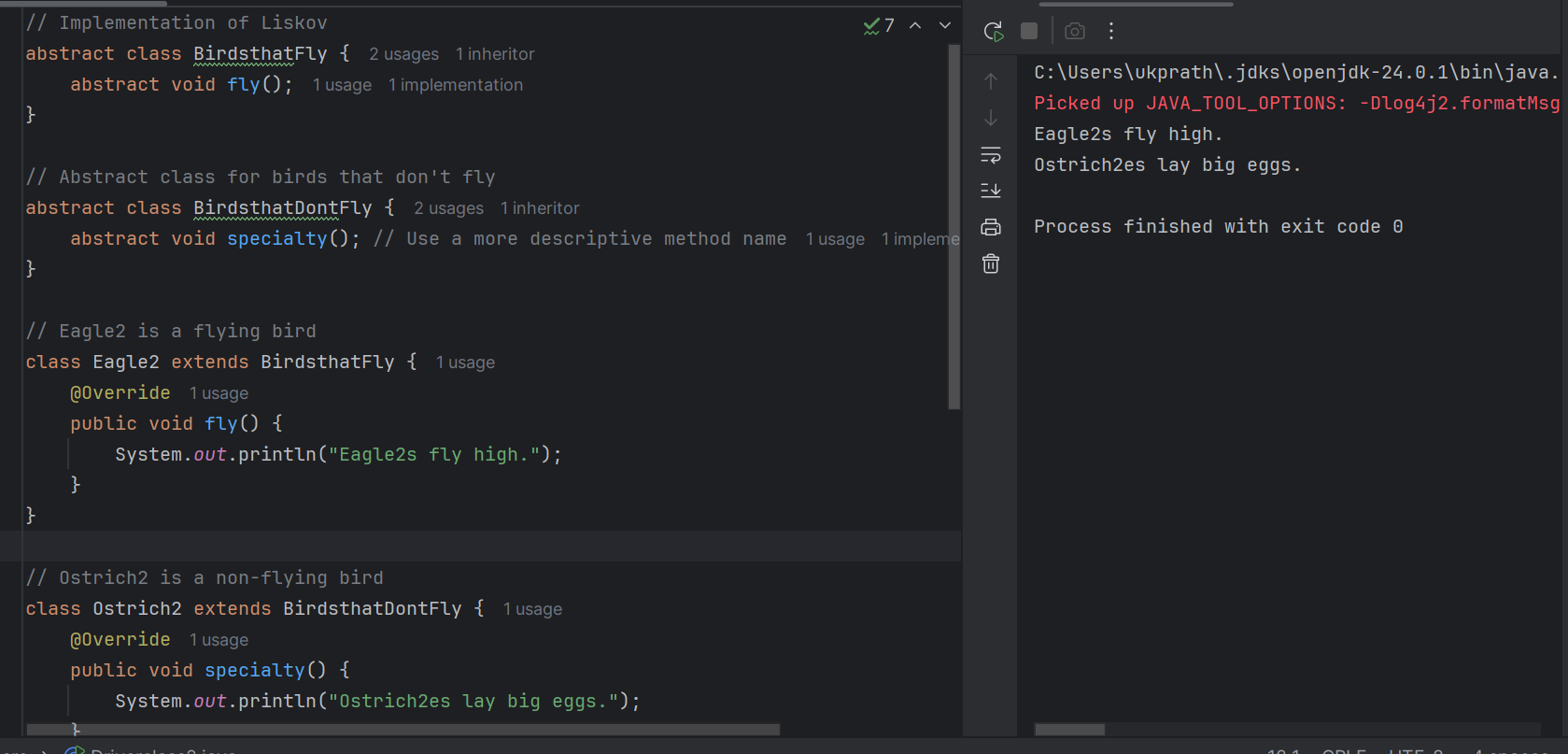
}

}



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( ,... ) {

}

}

interface ICalcShapesArea {

void calcArea();

void calcVolume(); // Problem: Not all shapes have volume

}

class Circle implements ICalcShapesArea {

@Override

public void calcArea() {

System.*out*.println("Calculating area of Circle...");

}

@Override

public void calcVolume() {

// ISP Violation: Circle has no volume, but must implement this

System.*out*.println("Circle has no volume! (dummy method)");

}

}

class Sphere implements ICalcShapesArea {

@Override

public void calcArea() {

System.*out*.println("Calculating surface area of Sphere...");

}

@Override

public void calcVolume() {

System.*out*.println("Calculating volume of Sphere...");

}

}

public class DriverClass2 {

public static void main(String[] args) {

ICalcShapesArea circle = new Circle();

circle.calcArea();

circle.calcVolume(); // Unnecessary and misleading

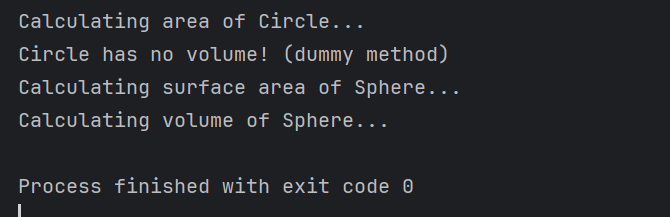
ICalcShapesArea sphere = new Sphere();

sphere.calcArea();

sphere.calcVolume();

}

}



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( ,... ) {

}

}

// Interface for shapes that have area and perimeter (2D shapes)

interface ICalcArea {

void calcArea();

void calcPerimeter();

}

// Interface for shapes that have volume (3D shapes)

interface ICalcVolume {

void calcVolume();

}

// Circle is a 2D shape: only implements ICalcArea

class Circle implements ICalcArea {

@Override

public void calcArea() {

System.*out*.println("Calculating area of Circle...");

}

@Override

public void calcPerimeter() {

System.*out*.println("Calculating perimeter of Circle...");

}

}

// Sphere3 is a 3D shape: implements both ICalcArea and ICalcVolume

class Sphere3 implements ICalcArea, ICalcVolume {

@Override

public void calcArea() {

System.*out*.println("Calculating surface area of Sphere3...");

}

@Override

public void calcPerimeter() {

System.*out*.println("Calculating circular cross-section perimeter...");

}

@Override

public void calcVolume() {

System.*out*.println("Calculating volume of Sphere3...");

}

}

public class DriverClass3 {

public static void main(String[] args) {

ICalcArea circle = new Circle();

circle.calcArea();

circle.calcPerimeter();

Sphere3 Sphere3 = new Sphere3();

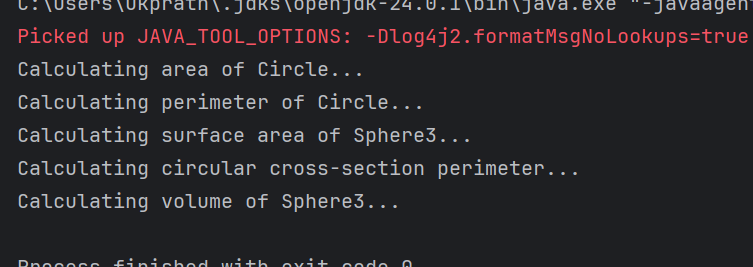
Sphere3.calcArea();

Sphere3.calcPerimeter();

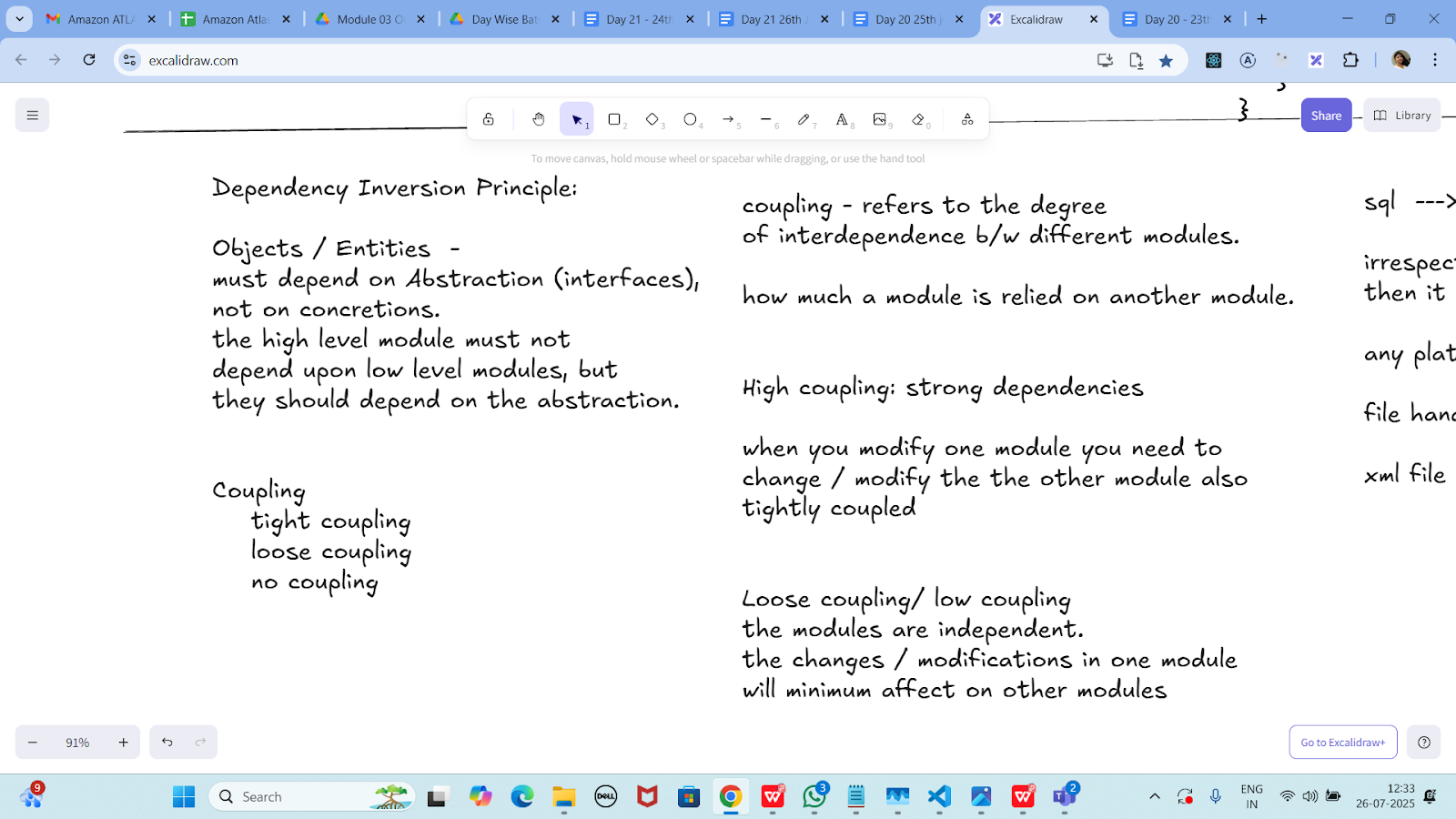
Sphere3.calcVolume();

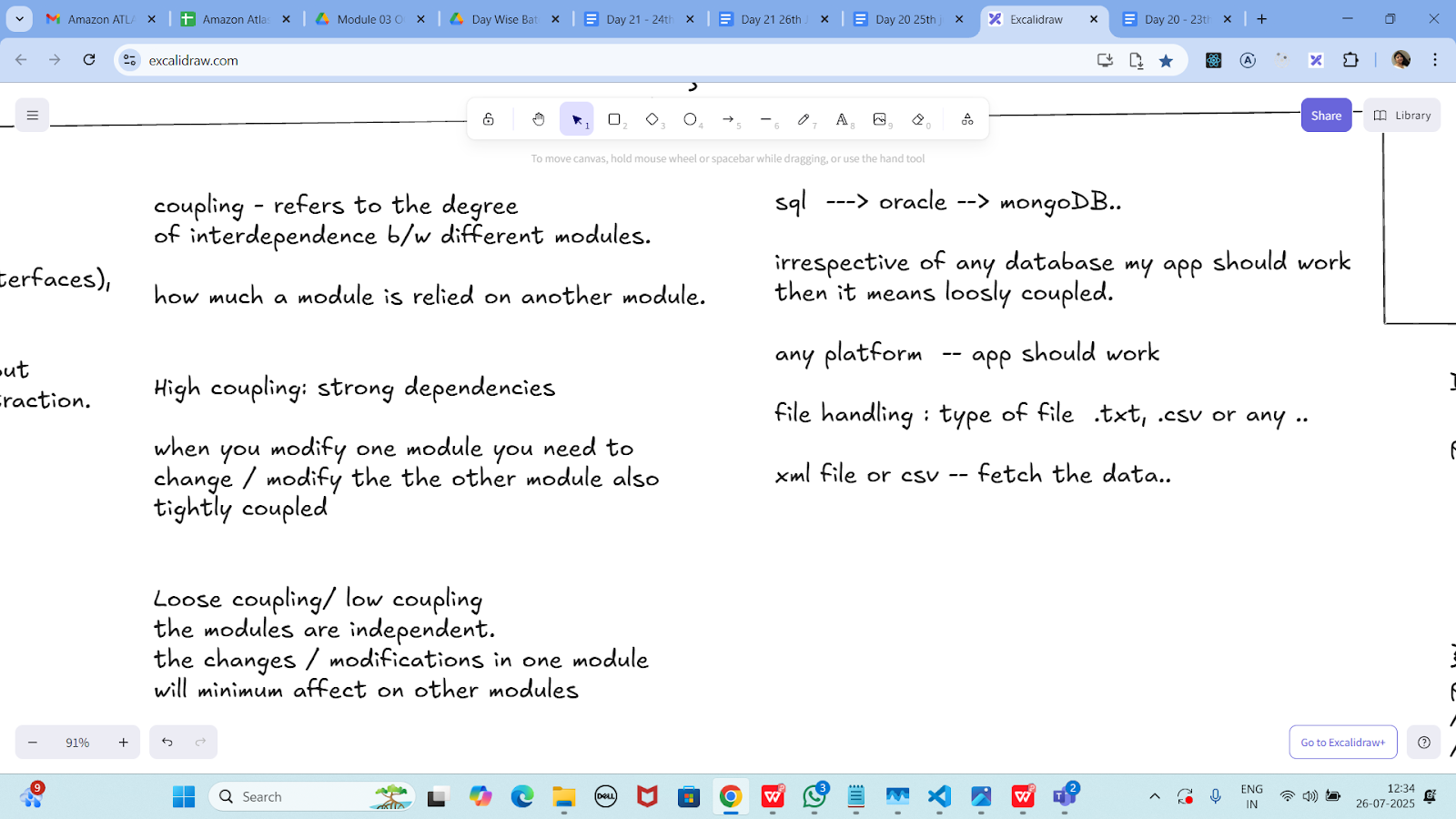
}

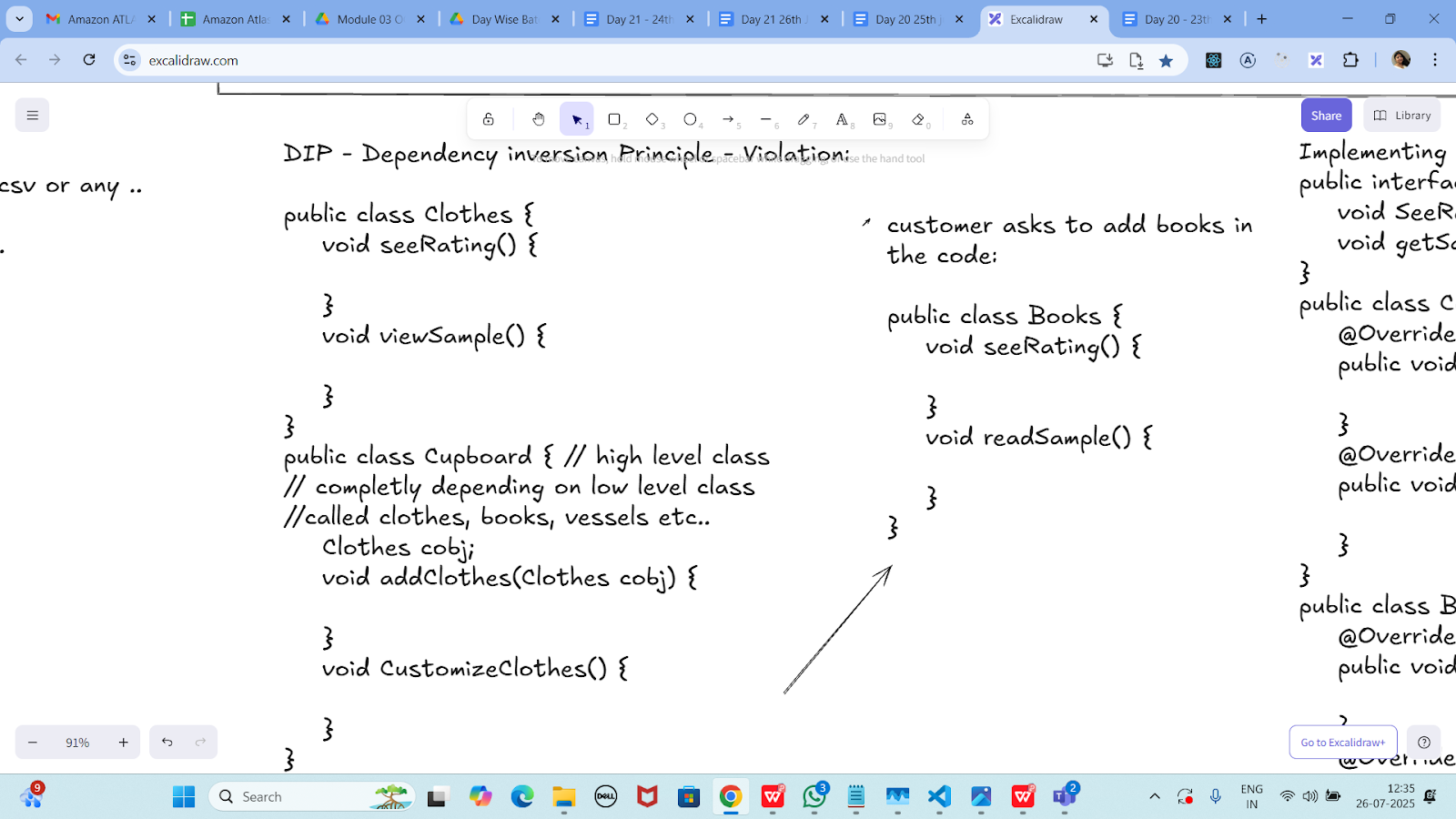
}



Dependency Inversion Principle:







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() {

}

}

// Low-level module

class Clothes {

void seeRating() {

System.*out*.println("Clothes rating shown");

}

void viewSample() {

System.*out*.println("Viewing clothes sample");

}

}

// Another low-level module (new customer request)

class Books {

void seeRating() {

System.*out*.println("Book rating shown");

}

void readSample() {

System.*out*.println("Reading book sample");

}

}

// High-level module that violates DIP

class Cupboard {

Clothes cobj;

Books bobj;

void addClothes(Clothes cobj) {

this.cobj = cobj;

}

void addBooks(Books bobj) {

this.bobj = bobj;

}

void customizeClothes() {

System.*out*.println("Customizing clothes:");

cobj.viewSample();

cobj.seeRating();

}

void customizeBooks() {

System.*out*.println("Customizing books:");

bobj.readSample();

bobj.seeRating();

}

}

// Main class with main method

public class DriverClass4 {

public static void main(String[] args) {

Cupboard cupboard = new Cupboard();

// Adding and customizing clothes

Clothes clothes = new Clothes();

cupboard.addClothes(clothes);

cupboard.customizeClothes();

// Adding and customizing books

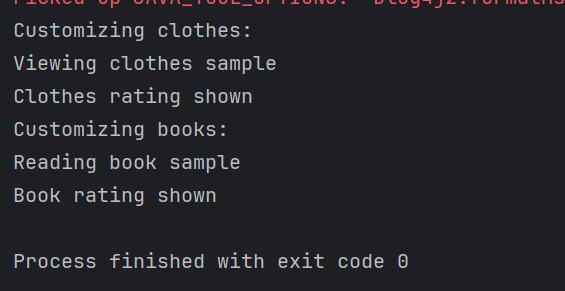
Books books = new Books();

cupboard.addBooks(books);

cupboard.customizeBooks();

}

}



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() {

}

}

interface IProduct {

void seeReviews();

void getSample();

}

class Clothes2 implements IProduct {

@Override

public void seeReviews() {

System.*out*.println("Showing reviews for Clothes2...");

}

@Override

public void getSample() {

System.*out*.println("Showing Clothes2 sample...");

}

}

class Books implements IProduct {

@Override

public void seeReviews() {

System.*out*.println("Showing reviews for Books...");

}

@Override

public void getSample() {

System.*out*.println("Reading book sample...");

}

}

class Cupboard2 {

private IProduct product;

public void addProduct(IProduct product) {

this.product = product;

}

public void customizeProduct() {

product.getSample();

product.seeReviews();

}

}

public class DriverClass5 {

public static void main(String[] args) {

Cupboard2 Cupboard2 = new Cupboard2();

IProduct Clothes2 = new Clothes2();

Cupboard2.addProduct(Clothes2);

System.*out*.println("=== Clothes2 ===");

Cupboard2.customizeProduct();

IProduct books = new Books();

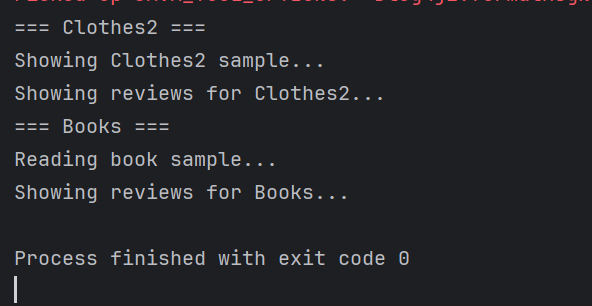
Cupboard2.addProduct(books);

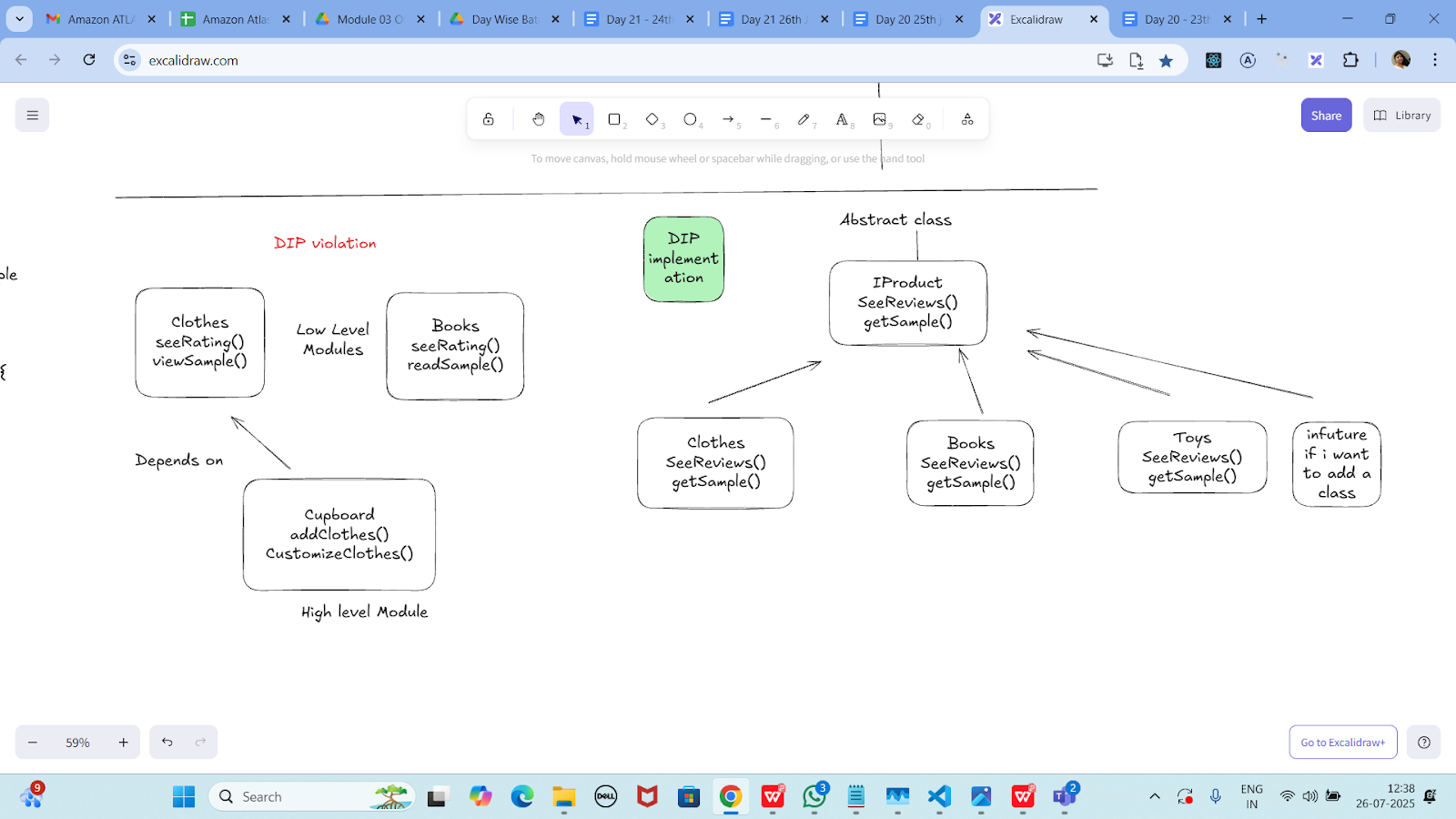
System.*out*.println("=== Books ===");

Cupboard2.customizeProduct();

}

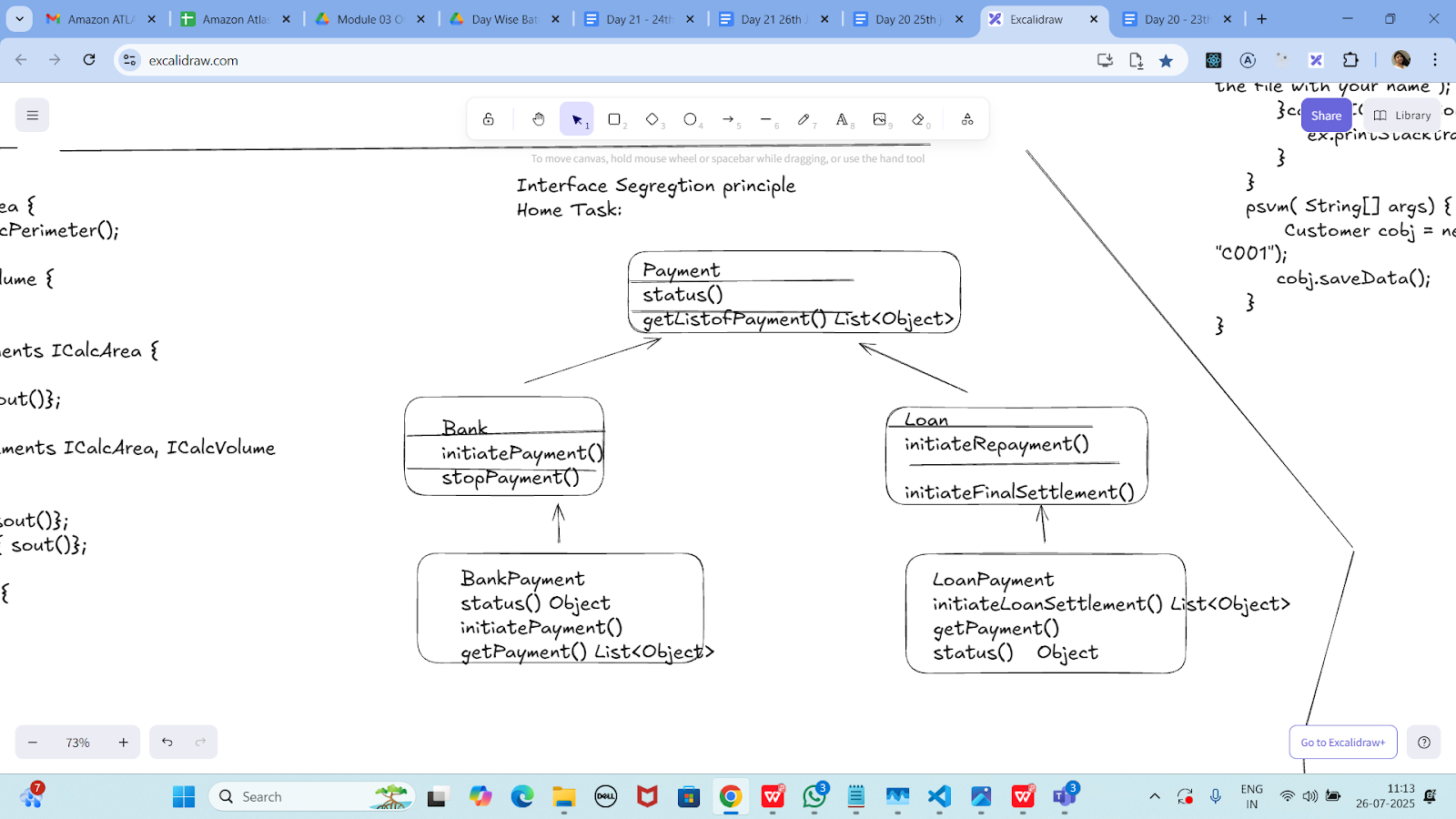
}





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

Home Tasks:



Task 02:

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 ,

wildcards:

1. unbounded Wildcard

? ===> any data type if you want to use you can use ?

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

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

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

===> you will use when ypou need accept a type and its super types

import java.util.ArrayList;

import java.util.List;

class Animal {

void sound() {

System.*out*.println("Sounds of different animals");

}

}

class Cat extends Animal {

@Override

void sound() {

System.*out*.println("Meow is the sound of cat");

}

}

public class hometask01 {

public static void main(String[] args) {

// Invariance: List<Cat> is NOT a subtype of List<Animal>

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

catList.add(new Cat());

// List<Animal> animals = catList; // Compile Error!

// Using unbounded wildcard

List<?> unknownList = catList;

Object obj = unknownList.get(0); // Allowed

System.*out*.println("Object from unknownList: " + obj.getClass().getSimpleName());

// Using upper bounded wildcard: List of unknown subtype of Animal

List<? extends Animal> animalsExtends = catList;

Animal a = animalsExtends.get(0); // Allowed to read as Animal

a.sound();

// animalsExtends.add(new Cat()); // Compile Error: can't add

// Using lower bounded wildcard: List of some supertype of Cat

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

animalsSuper.add(new Cat()); // Allowed to add Cat

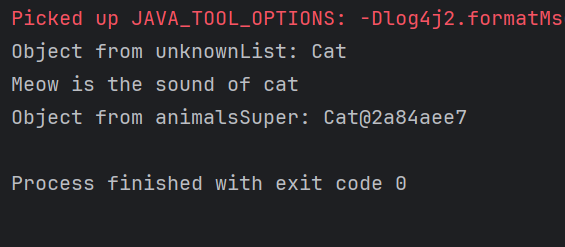
// Animal an = animalsSuper.get(0); // Compile Error: can't safely read as Animal

Object o = animalsSuper.get(0); // Can read as Object

System.*out*.println("Object from animalsSuper: " + o);

}

}



Unbounded wildcards:

Task 03:

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

void printList(List<?> list) {

for(Object element: list) {

sout (element);

}

}

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

clist.add(new Cat());

printList(clist); //

import java.util.ArrayList;

import java.util.List;

class Animal1 {

@Override

public String toString() {

return "Animal1";

}

}

class Cat1 extends Animal1 {

@Override

public String toString() {

return "Cat1";

}

}

public class hometask02 {

// Method that accepts a list of any type using unbounded wildcard

static void printList(List<?> list) {

for (Object element : list) {

System.*out*.println(element);

}

}

public static void main(String[] args) {

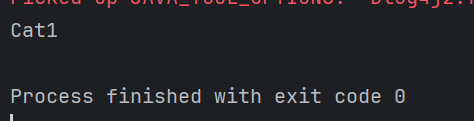
List<Cat1> Cat1List = new ArrayList<>();

Cat1List.add(new Cat1());

*printList*(Cat1List); // Prints: Cat1

}

}



---------------------------

Task 04:

Upper Bounded 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

import java.util.ArrayList;

import java.util.List;

class Animal3 {

void sound() {

System.*out*.println("Sounds of different Animal3s");

}

}

class Cat3 extends Animal3 {

@Override

void sound() {

System.*out*.println("Meow is the sound of Cat3");

}

}

public class hometask01 {

public static void main(String[] args) {

// Invariance: List<Cat3> is NOT a subtype of List<Animal3>

List<Cat3> Cat3List = new ArrayList<>();

Cat3List.add(new Cat3());

// List<Animal3> Animal3s = Cat3List; // Compile Error!

// Using unbounded wildcard

List<?> unknownList = Cat3List;

Object obj = unknownList.get(0); // Allowed

System.*out*.println("Object from unknownList: " + obj.getClass().getSimpleName());

// Using upper bounded wildcard: List of unknown subtype of Animal3

List<? extends Animal3> Animal3sExtends = Cat3List;

Animal3 a = Animal3sExtends.get(0); // Allowed to read as Animal3

a.sound();

// Animal3sExtends.add(new Cat3()); // Compile Error: can't add

// Using lower bounded wildcard: List of some supertype of Cat3

List<? super Cat3> Animal3sSuper = new ArrayList<Animal3>();

Animal3sSuper.add(new Cat3()); // Allowed to add Cat3

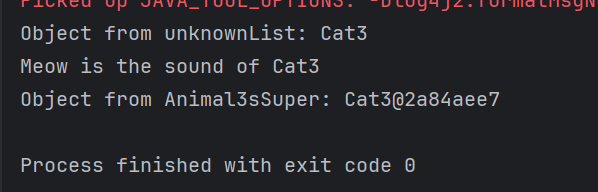
// Animal3 an = Animal3sSuper.get(0); // Compile Error: can't safely read as Animal3

Object o = Animal3sSuper.get(0); // Can read as Object

System.*out*.println("Object from Animal3sSuper: " + o);

}

}



----------------------------

Task 05:

lower Bounded Wildcards

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

cats.add(new Cat());

}

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

addAcat(animals); //

import java.util.ArrayList;

import java.util.List;

class Animal4 {

void sound() {

System.*out*.println("Animal4 makes a sound");

}

}

class Cat4 extends Animal4 {

@Override

void sound() {

System.*out*.println("Meow is the sound of Cat4");

}

}

public class hometask04 {

// Lower bounded wildcard: accepts Cat4 or any superclass (Animal4, Object)

static void addACat4(List<? super Cat4> Cat4s) {

Cat4s.add(new Cat4()); // ✅ Safe to add

System.*out*.println("Cat4 added to the list.");

}

public static void main(String[] args) {

List<Animal4> Animal4s = new ArrayList<>();

*addACat4*(Animal4s); // ✅ Works because Animal4 is a supertype of Cat4

// Show what's inside the list

for (Animal4 Animal4 : Animal4s) {

Animal4.sound(); // Output: Meow is the sound of Cat4

}

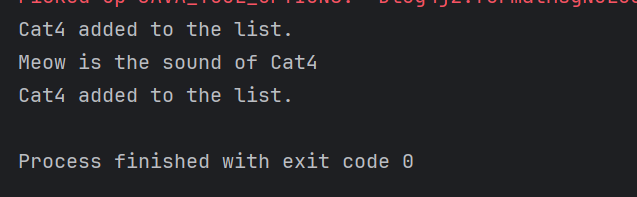
// Also works with List<Object>

List<Object> objects = new ArrayList<>();

*addACat4*(objects); // ✅ Still valid: Object is a supertype of Cat4

}

}



—----------------------------

Task 06:

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();

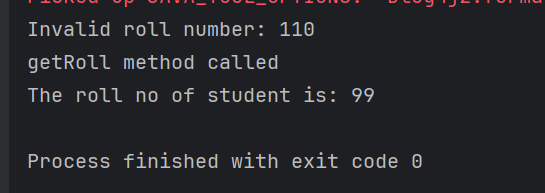
sobj.roll\_no = 10;

//sobj.roll\_no = 110;

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

}

}



Task 07:

// loose coupling

class Student {

private int roll\_no = 0;

public int getRoll() {

sout("getRoll method");

return roll\_no;

}

public void setRoll(int roll) {

if(!(roll > 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("the roll no of student is "+ sobj.getroll();

}

}

class Student2 {

private int roll\_no = 0;

public int getRoll() {

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

return roll\_no;

}

public void setRoll(int roll) {

if (!(roll > 100)) {

roll\_no = roll;

} else {

System.*out*.println("Invalid roll number: " + roll);

}

}

}

public class Loose\_Coupling {

public static void main(String[] args) {

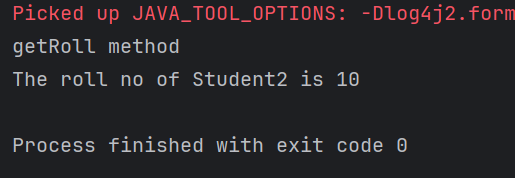
Student2 sobj = new Student2(); // Could also be: Person pobj = new Student2(); (with an interface)

sobj.setRoll(10); // Valid roll number

System.*out*.println("The roll no of Student2 is " + sobj.getRoll());

}

}



Task 08:

DIP - violation:

class LightBulb {

void turnOn() {

sout("light turned on");

}

void turnOff() {

sout("light is off");

}

}

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();

}

}

class LightBulb {

void turnOn() {

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

}

void turnOff() {

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

}

}

// DIP Violation: Switch directly depends on LightBulb

class Switch {

LightBulb lbulbobj;

Switch(LightBulb lbulbobj) {

this.lbulbobj = lbulbobj;

}

void operate() {

lbulbobj.turnOn();

}

public static void main(String[] args) {

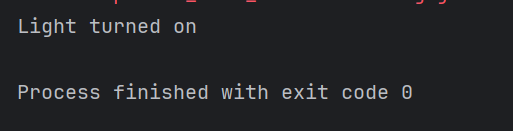
LightBulb lbulbobj = new LightBulb(); // tightly coupled

Switch switchObj = new Switch(lbulbobj);

switchObj.operate();

}

}



Task 09:

DIP implementation:

interface SwitchOnOff {

void turnOn();

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

}

class LightBulb {

void turnOn() {

sout("light turned on");

}

void turnOff() {

sout("light is off");

}

} // 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();

}

}

class LightBulb {

void turnOn() {

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

}

void turnOff() {

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

}

}

// DIP Violation: Switcht2 directly depends on LightBulb

class Switcht2 {

LightBulb lbulbobj;

Switcht2(LightBulb lbulbobj) {

this.lbulbobj = lbulbobj;

}

void operate() {

lbulbobj.turnOn();

}

public static void main(String[] args) {

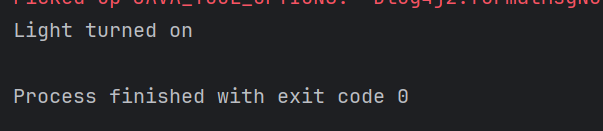
LightBulb lbulbobj = new LightBulb(); // tightly coupled

Switcht2 Switcht2Obj = new Switcht2(lbulbobj);

Switcht2Obj.operate();

}

}



Task 10

Refer the below doc in Docs to study

Doc 07 UML Class Diagrams

Dependency

class Engine {

void start() {

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

}

}

class Car {

void drive() {

Engine engine = new Engine(); // Dependency

engine.start();

System.*out*.println("Car is driving");

}

}

public class DependencyExample {

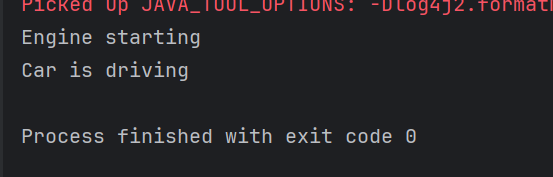
public static void main(String[] args) {

Car myCar = new Car();

myCar.drive();

}

}



Task 11:

Aggregation

class Driver {

private String name;

public Driver(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

class Car2 {

private Driver driver; // Aggregation

public Car2(Driver driver) {

this.driver = driver;

}

public Driver getDriver() {

return driver;

}

}

public class AggregationExample {

public static void main(String[] args) {

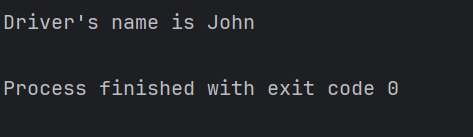
Driver driver = new Driver("John");

Car2 myCar2 = new Car2(driver);

System.*out*.println("Driver's name is " + myCar2.getDriver().getName());

}

}



Task 12:

Composition

class Wheel {

public Wheel() {

System.*out*.println("Wheel created");

}

}

class Car3 {

private Wheel[] wheels;

public Car3() {

// Car3 creates and owns wheels

wheels = new Wheel[4];

for (int i = 0; i < wheels.length; i++) {

wheels[i] = new Wheel();

}

}

}

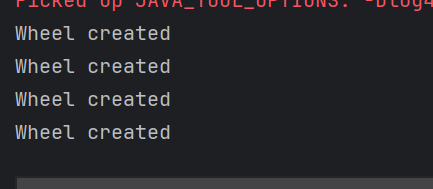
public class CompositionExample {

public static void main(String[] args) {

Car3 myCar3 = new Car3();

}

}



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

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

**Info box:**

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

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

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

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

Updated link: @ 12.32 26th July 2025 is the below one

<https://excalidraw.com/#json=j65iMkcyw7xx05kC40oPT,KweCYuCM4owgVnjHoV4Llw>

Generics

Wildcards characters

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

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