Day 21

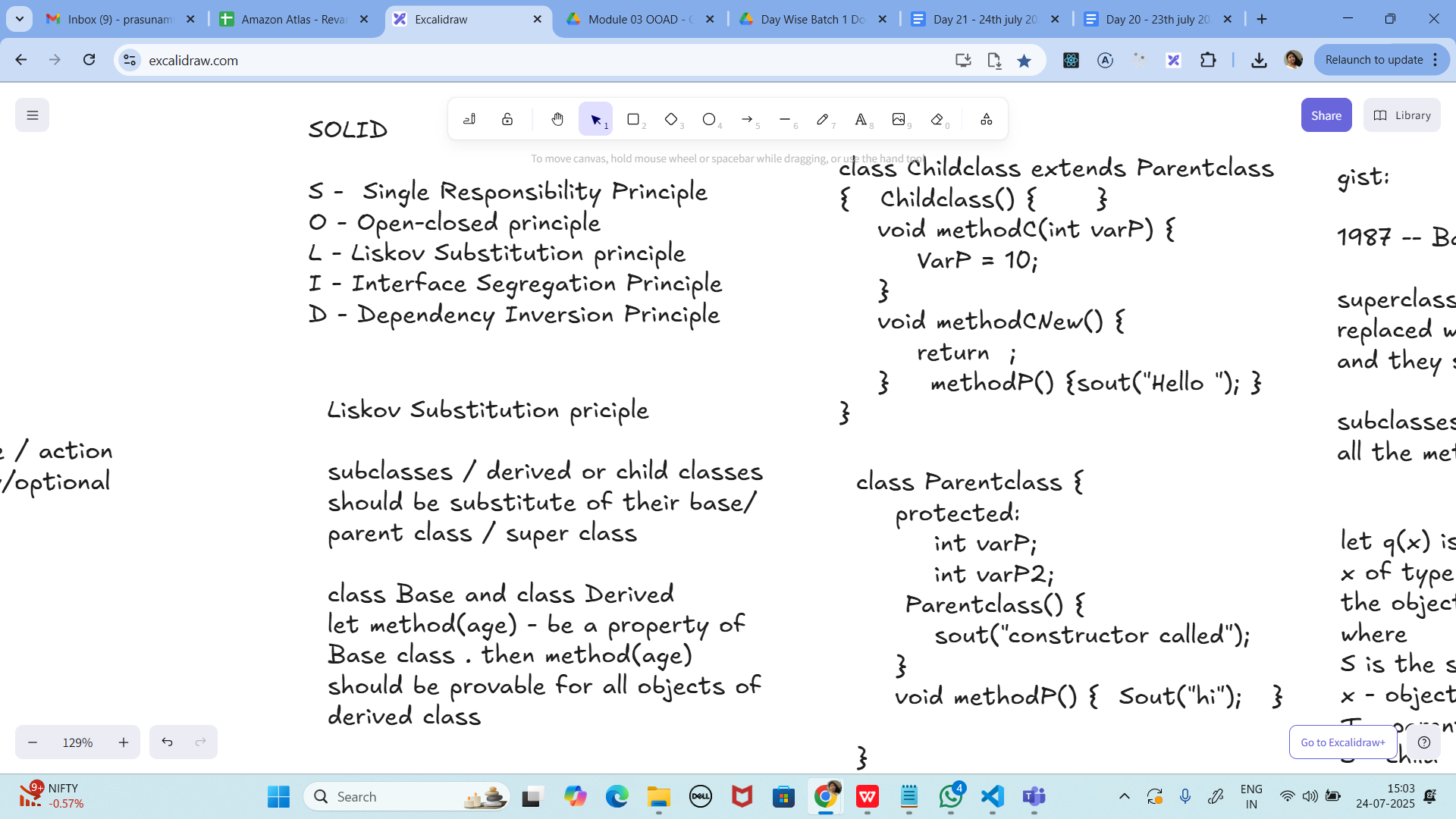
Employee ID: 201933938

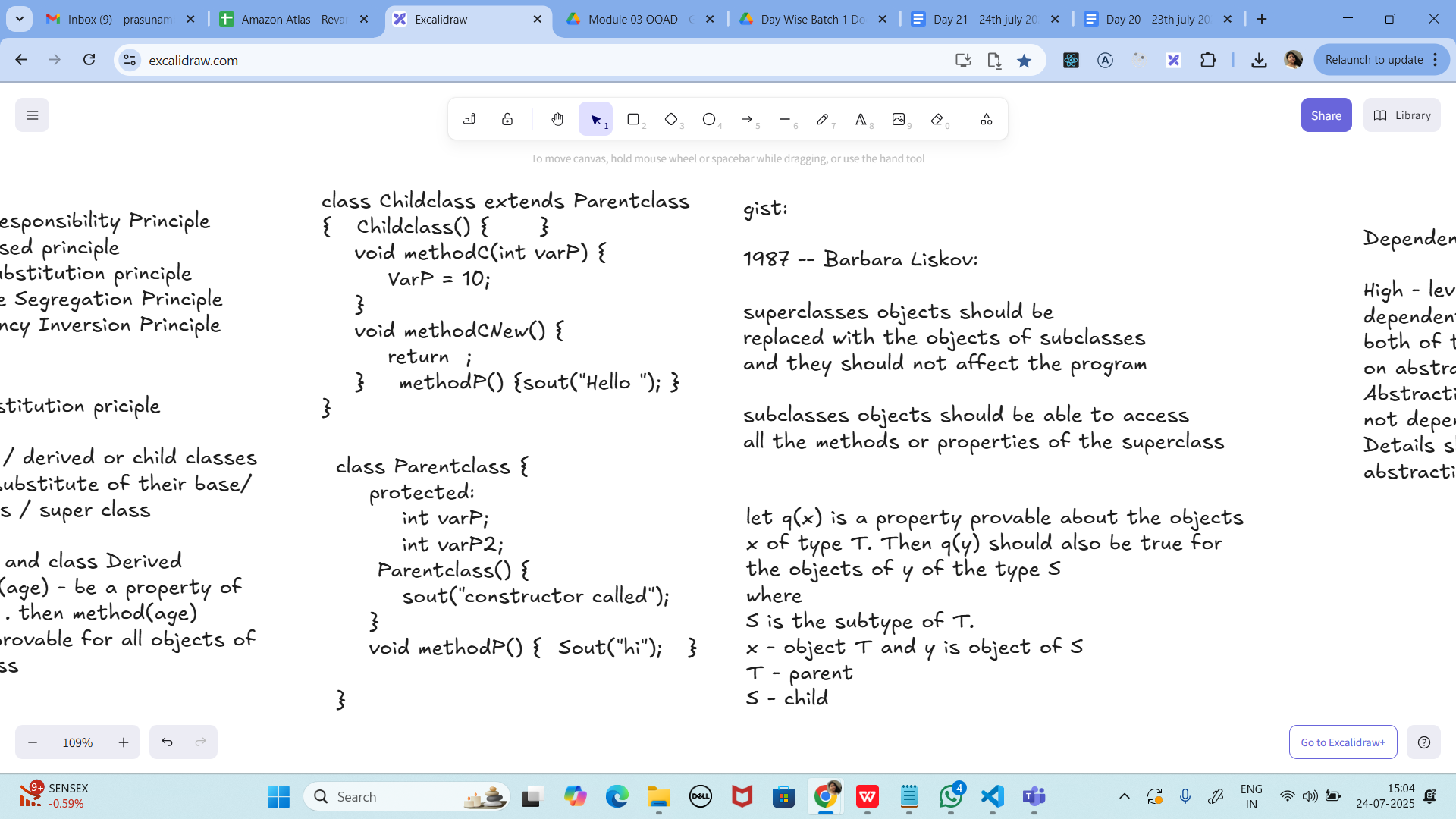
Login ID: iamasif

Name: Shaik Asif

SOLID

Liskov Substitution Principle 👍





Ta

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{

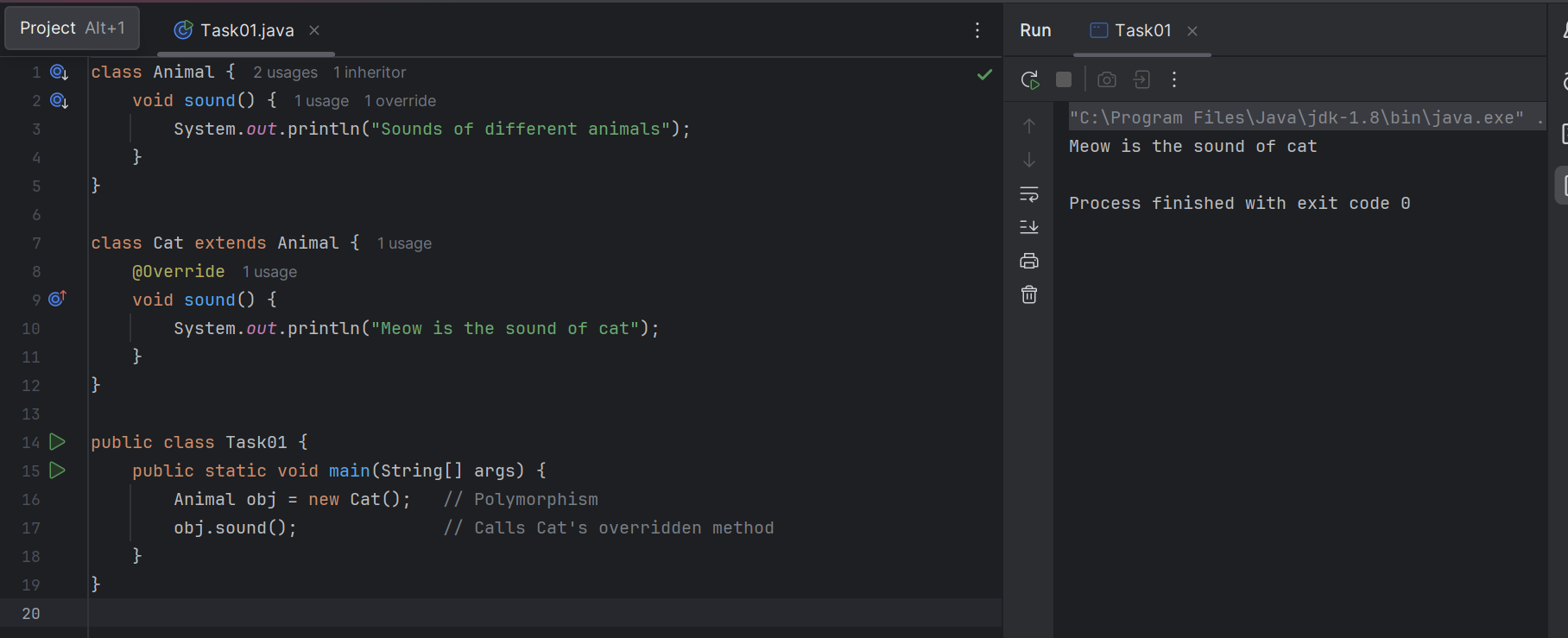
psvm(String[] args) {

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

Unbounded wildcards:

they are useful whern 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); //

Task 02: void printList(List<?> list) {  
     for(Object element: list) {  
       sout (element);    
     }   
   }

  List<Cat> clist = new ArrayList<>();  
   clist.add(new Cat());  
   printList(clist); //

**PROGRAM:**

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 Task01 {

// Method with unbounded wildcard

static void printList(List<?> list) {

for (Object element : list) {

System.out.println(element); // Prints element's toString()

}

}

public static void main(String[] args) {

// Polymorphism example

Animal obj = new Cat();

obj.sound(); // Calls Cat's overridden sound()

// Create a list of Cat objects

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

clist.add(new Cat());

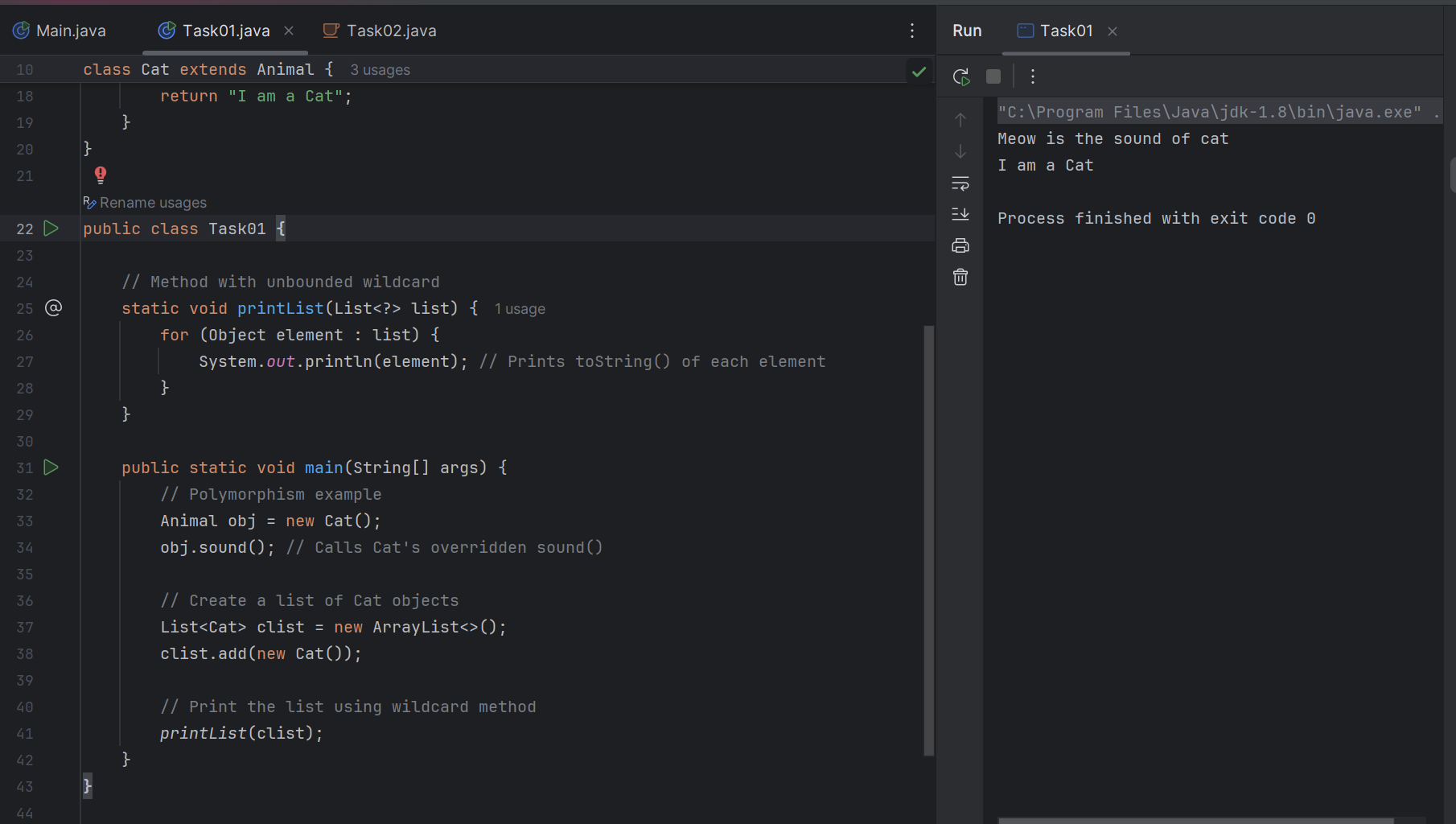
// Print the list using wildcard method

printList(clist);

}

}

**OUTPUT:**



.

Task 03:

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); //

**PROGRAM:**

**import java.util.ArrayList;**

**import java.util.List;**

**class Animal {**

**void sound() {**

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

**}**

**@Override**

**public String toString() {**

**return "I am an Animal";**

**}**

**}**

**class Cat extends Animal {**

**@Override**

**void sound() {**

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

**}**

**@Override**

**public String toString() {**

**return "I am a Cat";**

**}**

**}**

**class Dog extends Animal {**

**@Override**

**void sound() {**

**System.*out*.println("Woof is the sound of dog");**

**}**

**@Override**

**public String toString() {**

**return "I am a Dog";**

**}**

**}**

**public class Task01 {**

**// ✅ Unbounded wildcard example**

**static void printList(List<?> list) {**

**for (Object element : list) {**

**System.*out*.println(element); // Calls toString()**

**}**

**}**

**// ✅ Upper bounded wildcard example**

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

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

**element.sound(); // Safe, because it's at least an Animal**

**}**

**}**

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

**// Polymorphism example**

**Animal obj = new Cat();**

**obj.sound(); // Runtime polymorphism**

**// List of Cat**

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

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

**// Unbounded wildcard → prints toString()**

**System.*out*.println("\n-- printList(cats) --");**

***printList*(cats);**

**// Upper bounded wildcard → calls sound() for each**

**System.*out*.println("\n-- animalSound(cats) --");**

***animalSound*(cats);**

**// List of Dog (also works)**

**List<Dog> dogs = new ArrayList<>();**

**dogs.add(new Dog());**

**System.*out*.println("\n-- animalSound(dogs) --");**

***animalSound*(dogs);**

**// Mixed List<Animal>**

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

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

**animals.add(new Dog());**

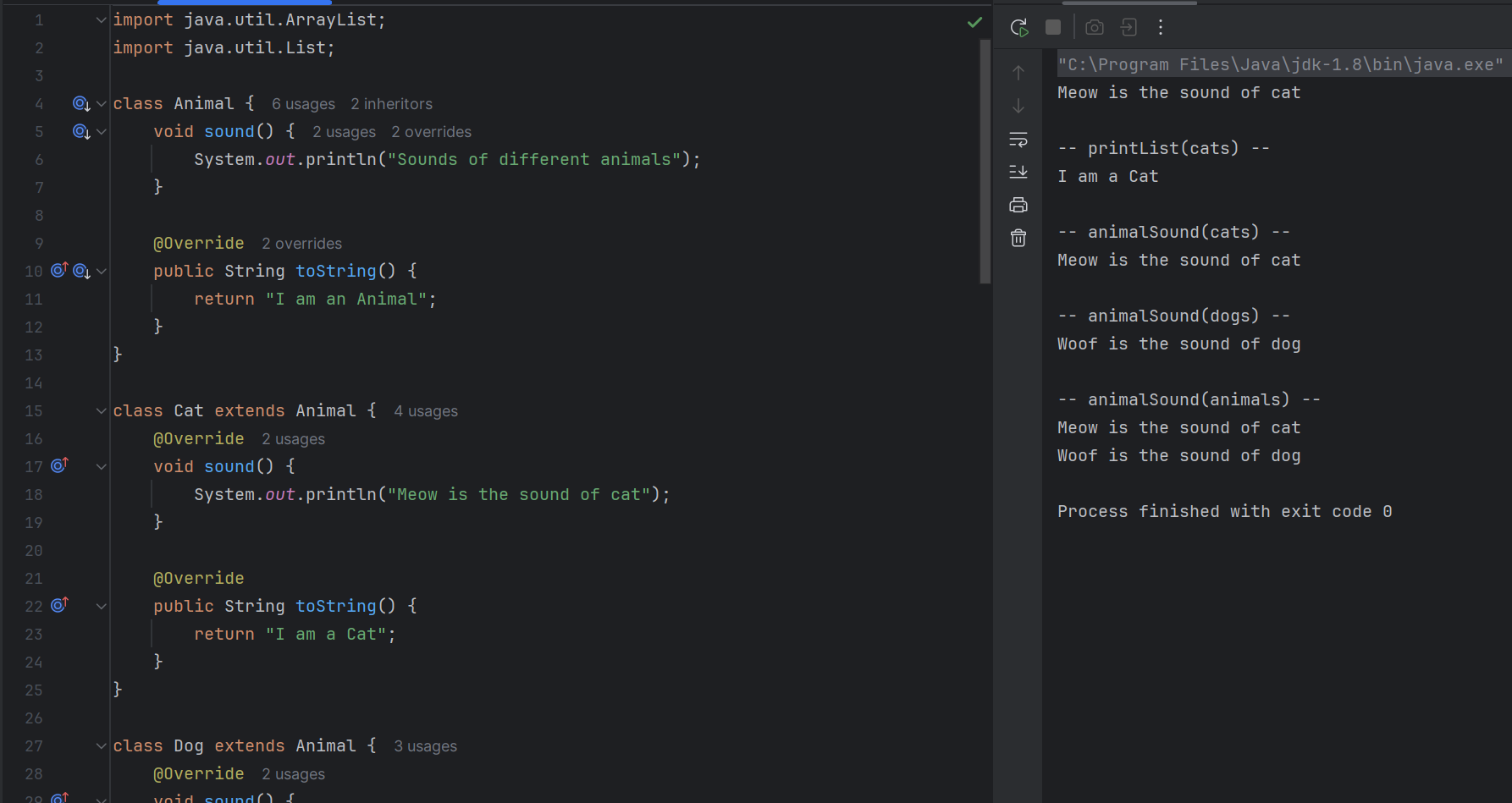
**System.*out*.println("\n-- animalSound(animals) --");**

***animalSound*(animals);**

**}**

**}**

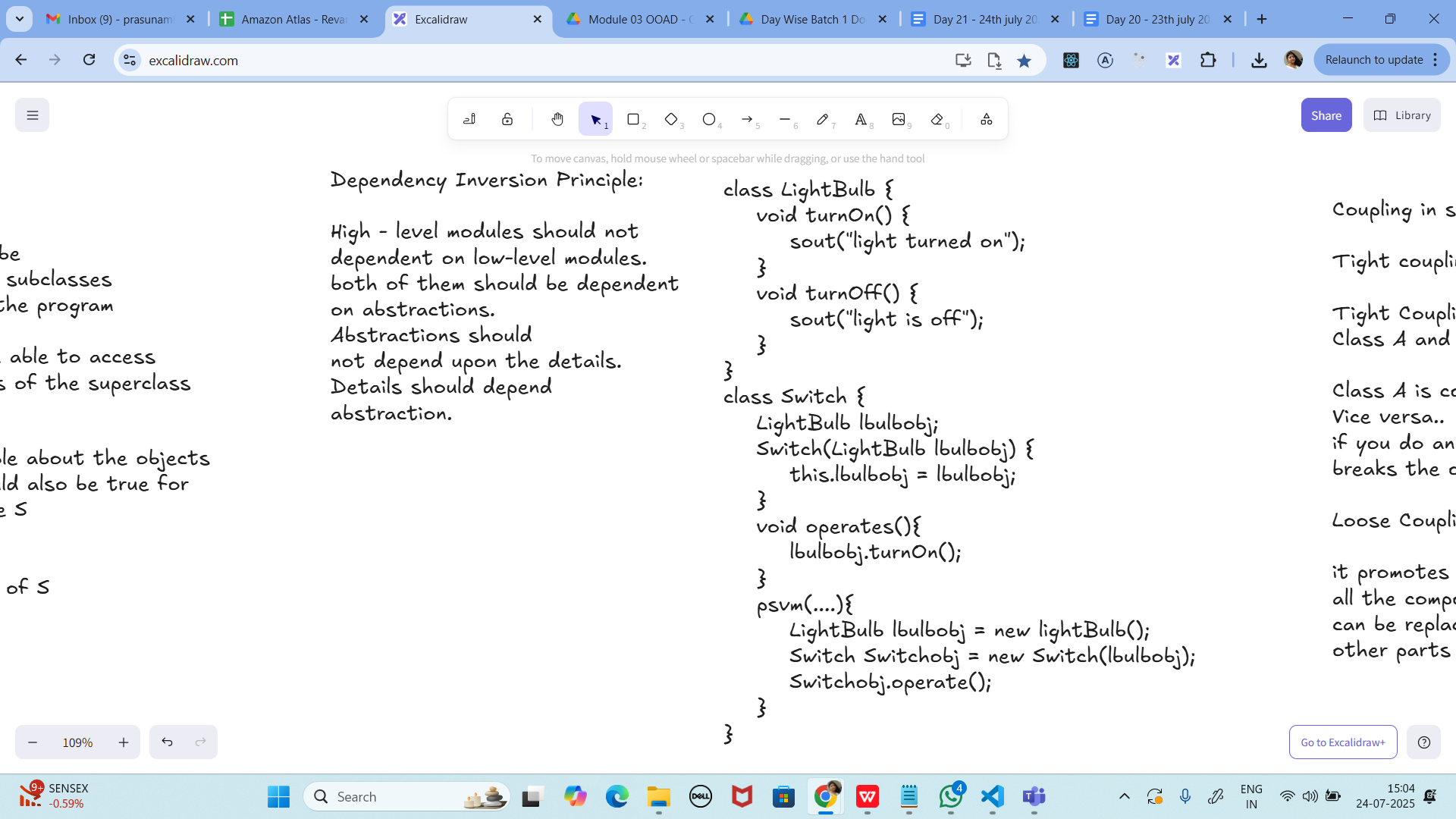
**OUTPUT:**

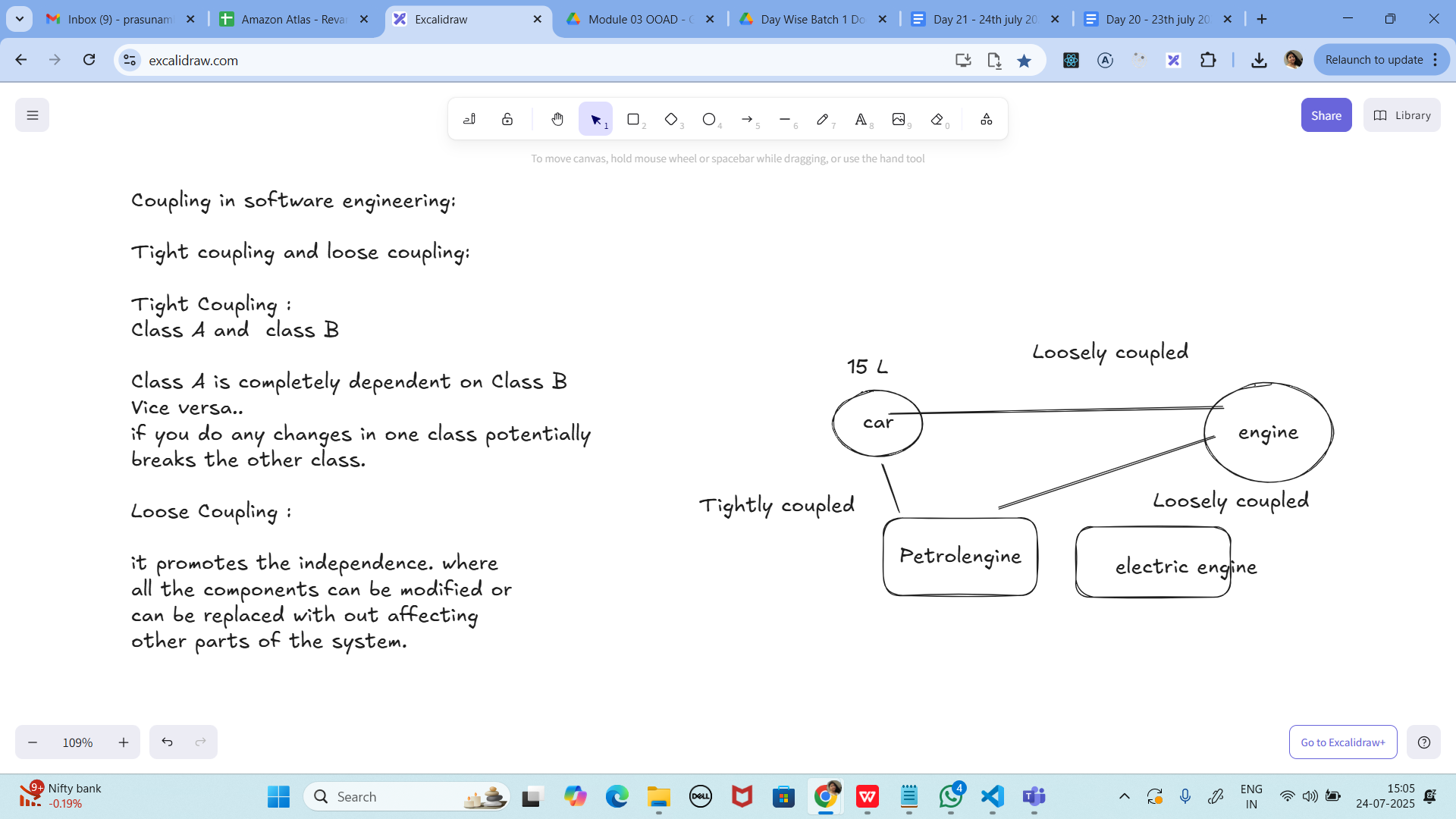


Task 04:

lower Bounded Wildcards

  void addAcat(List<? super Cat> cats) {  
     cats.add(new Cat());  
   }  
   List<Animal> animals = new ArrayList<>();  
   addAcat(animals); //





Task 01:

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

}

}

10.33 to 10.38

.

**PROGRAM:**

**import java.util.ArrayList;**

**import java.util.List;**

**class Animal {**

**void sound() {**

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

**}**

**@Override**

**public String toString() {**

**return "I am an Animal";**

**}**

**}**

**class Cat extends Animal {**

**@Override**

**void sound() {**

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

**}**

**@Override**

**public String toString() {**

**return "I am a Cat";**

**}**

**}**

**class Dog extends Animal {**

**@Override**

**void sound() {**

**System.*out*.println("Woof is the sound of dog");**

**}**

**@Override**

**public String toString() {**

**return "I am a Dog";**

**}**

**}**

**public class Task01 {**

**// Unbounded wildcard example**

**static void printList(List<?> list) {**

**for (Object element : list) {**

**System.*out*.println(element);**

**}**

**}**

**// Upper bounded wildcard example**

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

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

**element.sound();**

**}**

**}**

**// Lower bounded wildcard example**

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

**cats.add(new Cat()); // Safe because it's Cat or its supertype**

**System.*out*.println("A Cat has been added to the list.");**

**}**

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

**// Polymorphism example**

**Animal obj = new Cat();**

**obj.sound(); // Runtime polymorphism**

**// List of Cat**

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

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

**// Unbounded wildcard → prints toString()**

**System.*out*.println("\n-- printList(cats) --");**

***printList*(cats);**

**// Upper bounded wildcard → calls sound() for each**

**System.*out*.println("\n-- animalSound(cats) --");**

***animalSound*(cats);**

**// List of Dog**

**List<Dog> dogs = new ArrayList<>();**

**dogs.add(new Dog());**

**System.*out*.println("\n-- animalSound(dogs) --");**

***animalSound*(dogs);**

**// Mixed List<Animal>**

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

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

**animals.add(new Dog());**

**System.*out*.println("\n-- animalSound(animals) --");**

***animalSound*(animals);**

**// Lower bounded wildcard demo**

**System.*out*.println("\n-- addACat(animals) --");**

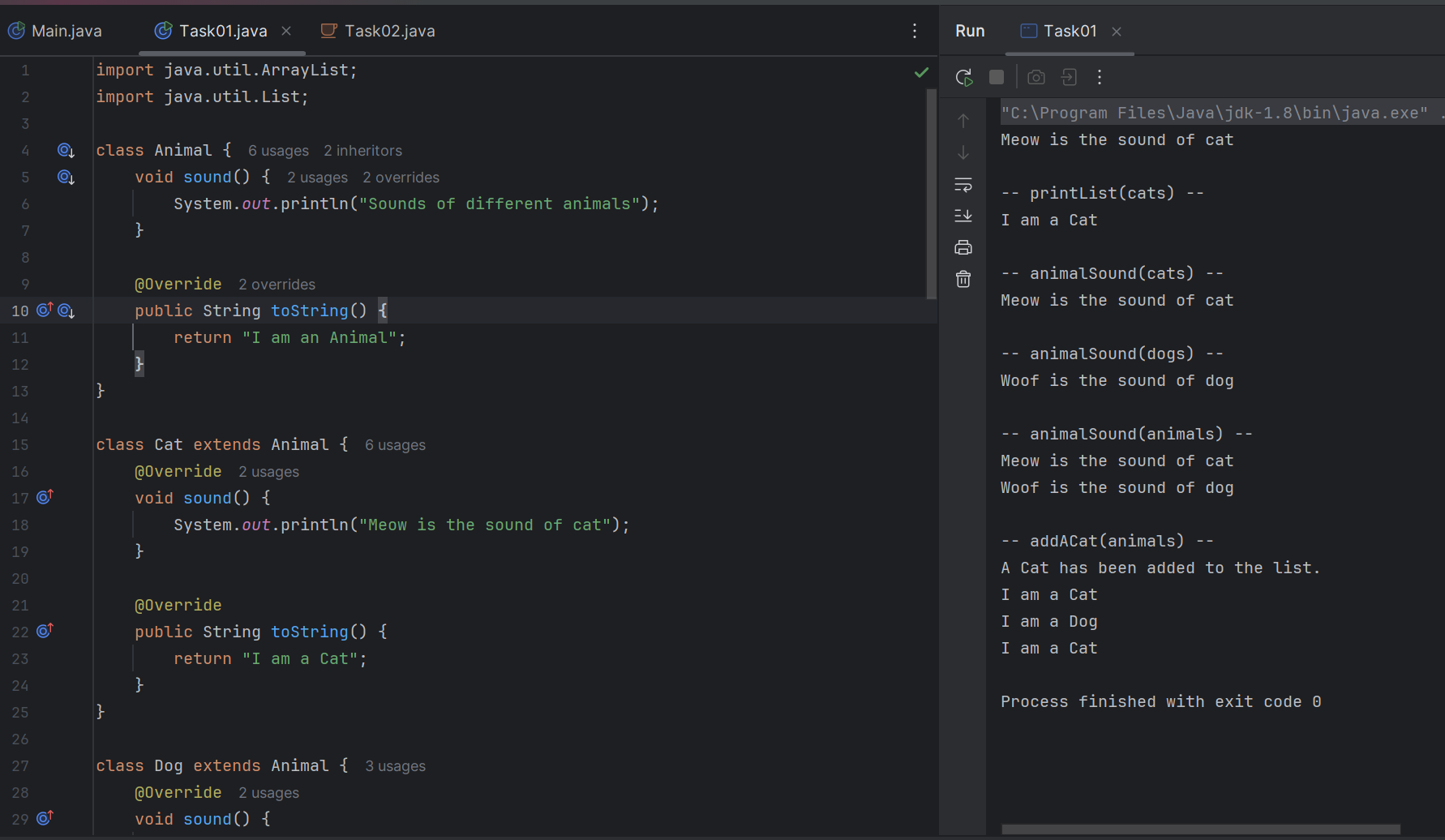
***addACat*(animals); // Adds a new Cat to animals**

***printList*(animals); // Now animals list contains 3 elements**

**}**

**}**

**OUTPUT:**

****

task5:

Tight coupling code class Student {

  public int roll\_no = 10;

  //private int roll\_no = 10;

  public int getRoll() {

    sout("getRoll method");

    return roll\_no;

  }

  public void setRoll(int roll) {

    if(!(roll > 100)

      roll\_no = roll;

  }

}

class Tight\_coupling {

  psvm(String....) {

    Student sobj = new Student();

    sobj.roll\_no = 10;

    //sobj.roll\_no = 110;

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

  }

}

**PROGRAM:**

**class Student {**

**// Keep it private to protect data**

**private int rollNo = 10;**

**public int getRoll() {**

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

**return rollNo;**

**}**

**public void setRoll(int roll) {**

**if (roll > 0 && roll <= 100) { // Validation logic**

**this.rollNo = roll;**

**} else {**

**System.*out*.println("Invalid roll number! Must be between 1 and 100.");**

**}**

**}**

**}**

**public class Task3 {**

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

**Student student = new Student();**

**// Tight coupling way (direct field access)**

**// student.roll\_no = 110; // Not allowed now (field is private)**

**// Proper way using setter (loose coupling)**

**student.setRoll(50); // Valid**

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

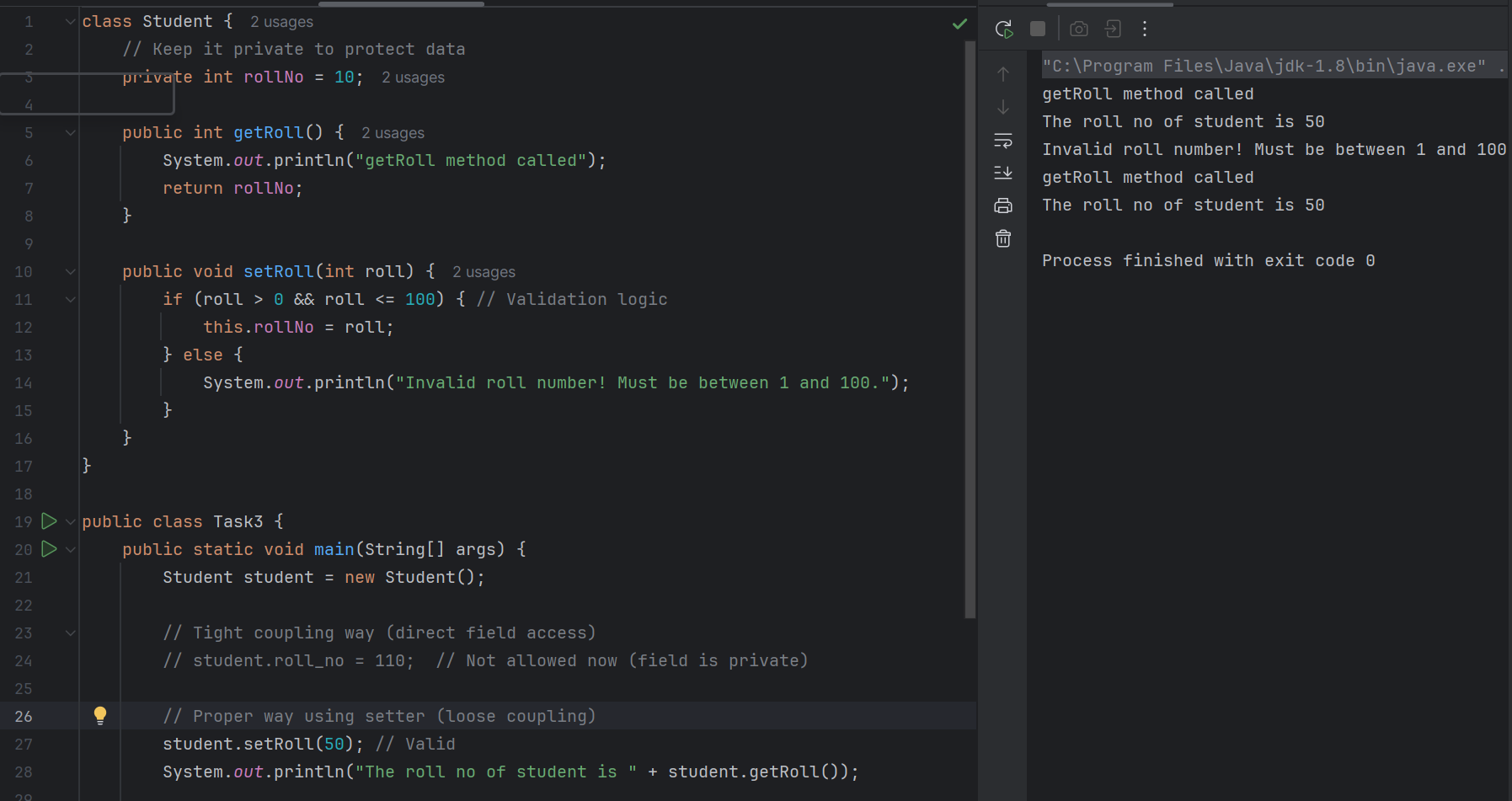
**student.setRoll(150); // Invalid**

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

**}**

**}**

**OUTPUT:**

****

Task 06:

// 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();

}

}

/\*

Module 1 Module 2 Module 3

independent .. they do communicate

// we should do coupling -- importance

good maintainability

testability

Error propagation

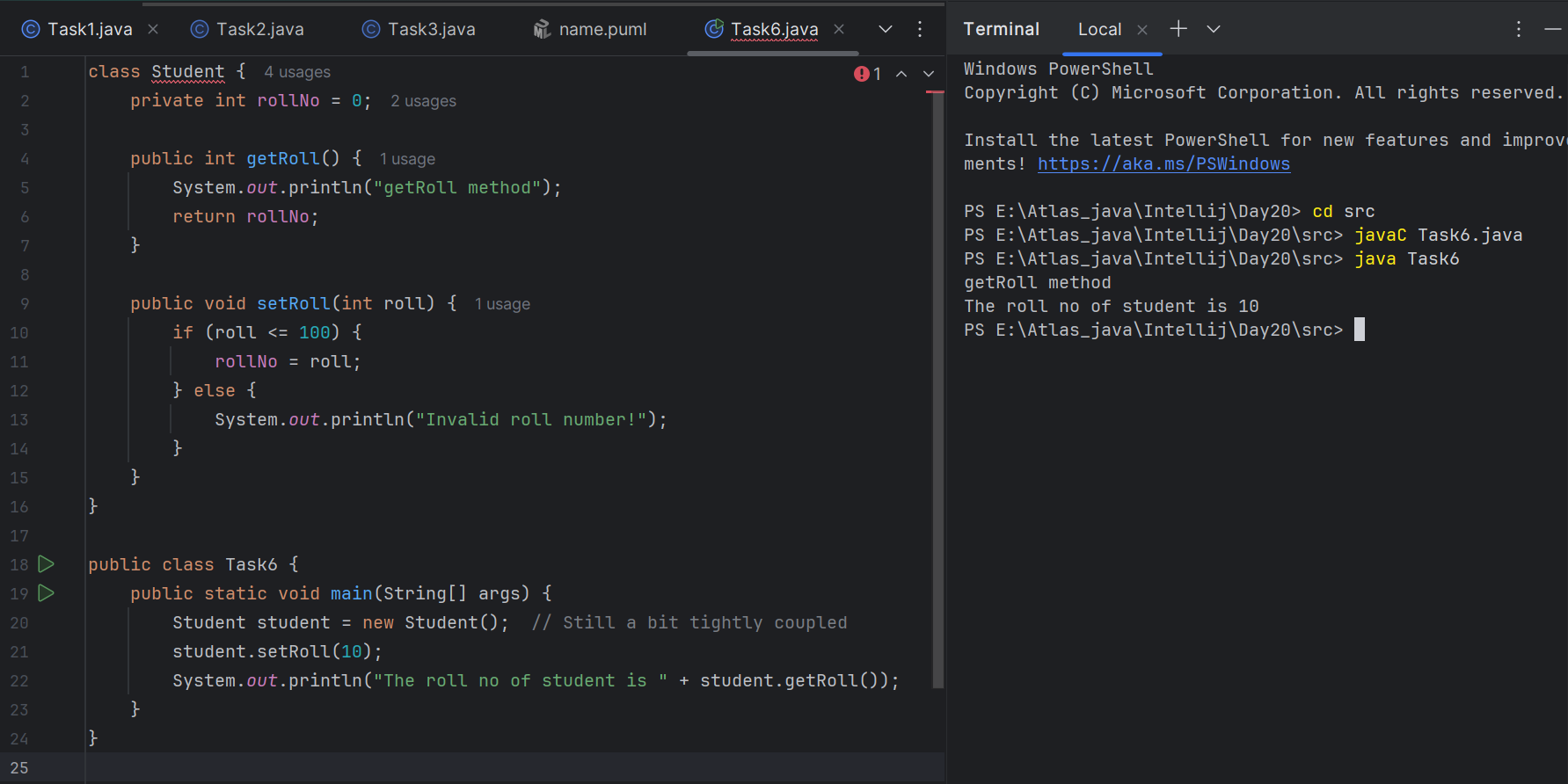
high coupling - errors and it will propagate throught the project.. not at all recommandable..

reusability:

No coupling :

the modules wont interact with eachother.. -- completely independent..

\*/



Task 07:

//DIP violating

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(....){

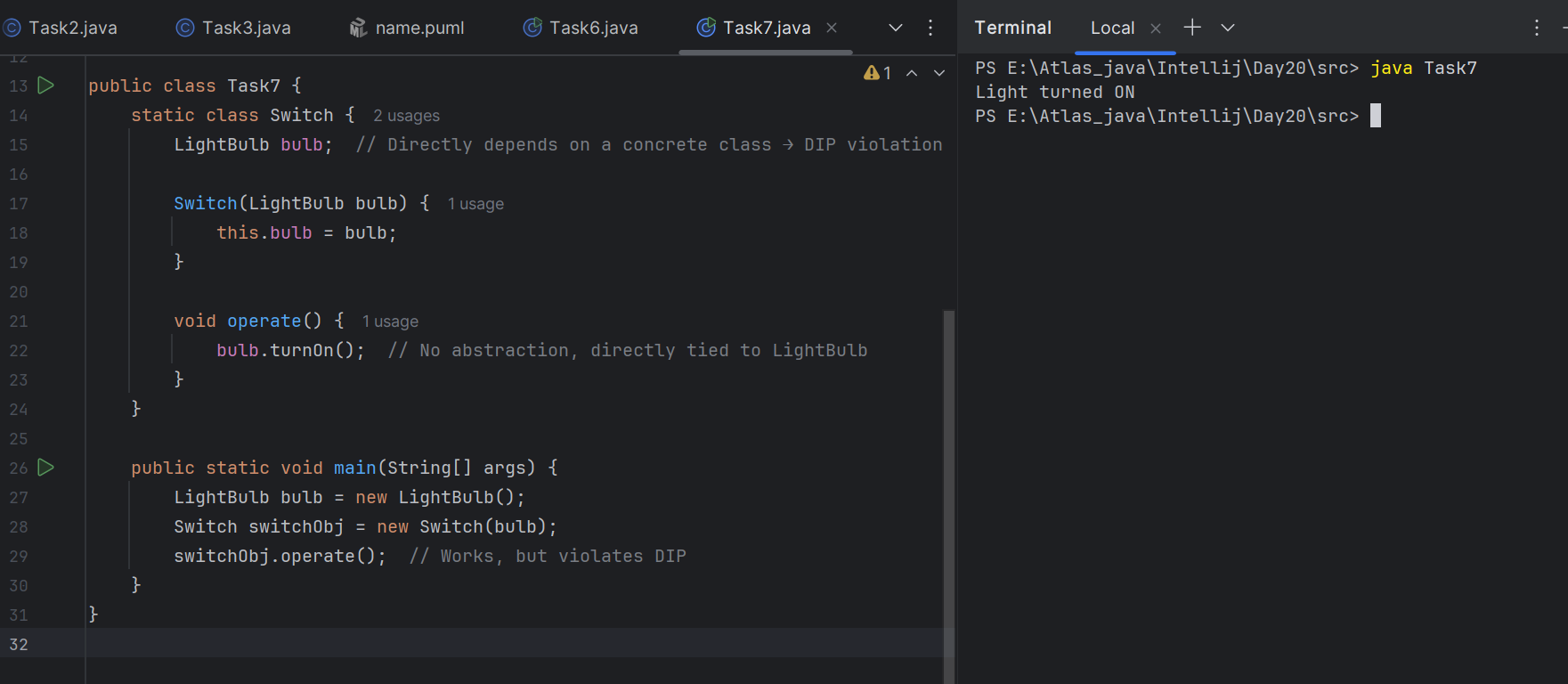
LightBulb lbulbobj = new lightBulb();

Switch Switchobj = new Switch(lbulbobj);

Switchobj.operate();

}

}



Task 08:

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(....){

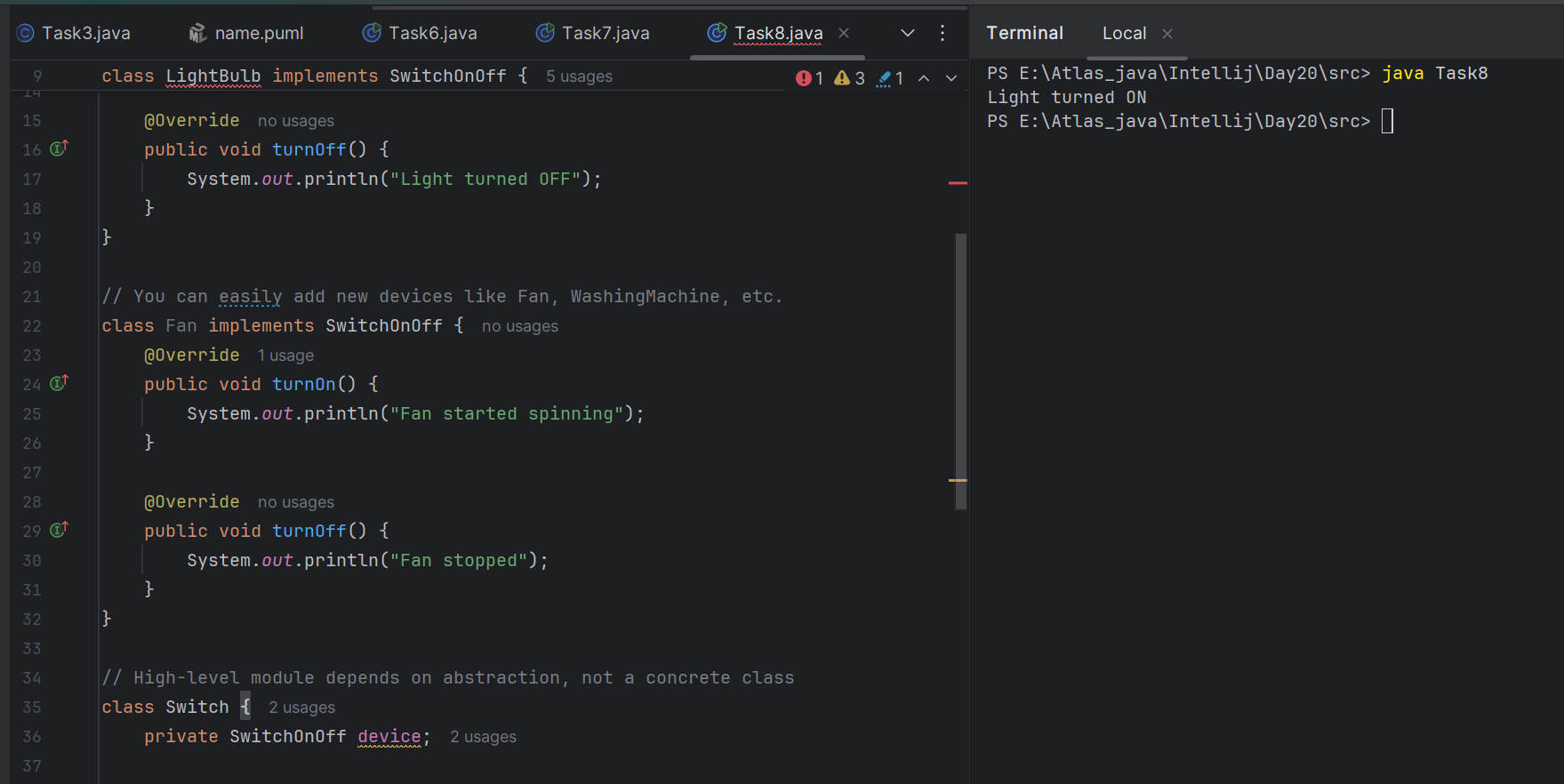
SwitchOnOff lbulbobj = new LightBulb();

Switch lightswitch = new Switch(lbulbobj);

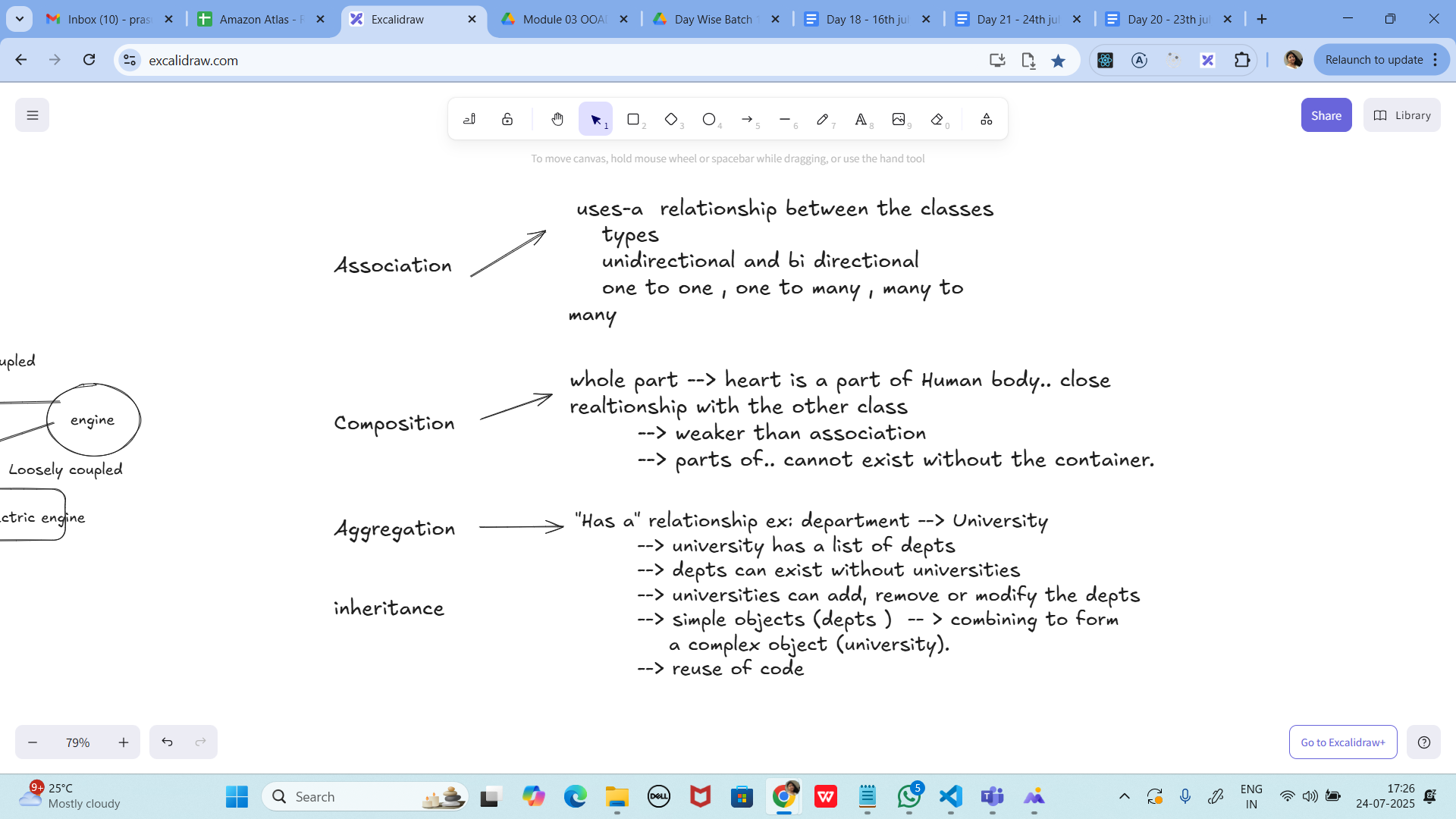
lightswitch.operate();

}

}



Association, Composition, Aggregation, Inheritance



**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();**

**}**

**}**

**DIP implementation.. in the above code**

**PROGRAM:**

**// Abstraction (High-level module depends on this, not on concrete classes)**

**interface SwitchOnOff {**

**void turnOn();**

**void turnOff();**

**}**

**// Concrete implementation (Low-level module)**

**class LightBulb implements SwitchOnOff {**

**@Override**

**public void turnOn() {**

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

**}**

**@Override**

**public void turnOff() {**

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

**}**

**}**

**// Another possible implementation (for future extension)**

**class Fan implements SwitchOnOff {**

**@Override**

**public void turnOn() {**

**System.*out*.println("Fan started spinning");**

**}**

**@Override**

**public void turnOff() {**

**System.*out*.println("Fan stopped");**

**}**

**}**

**// High-level module depends on abstraction, NOT concrete LightBulb**

**class Switch {**

**private SwitchOnOff device;**

**// Constructor Injection**

**public Switch(SwitchOnOff device) {**

**this.device = device;**

**}**

**public void operate() {**

**device.turnOn(); // only depends on interface**

**}**

**}**

**// Main class demonstrating DIP**

**public class Task04 {**

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

**// We can easily swap devices without modifying Switch class**

**SwitchOnOff bulb = new LightBulb();**

**Switch lightSwitch = new Switch(bulb);**

**lightSwitch.operate(); // Output: Light turned ON**

**// Future extension: Fan**

**SwitchOnOff fan = new Fan();**

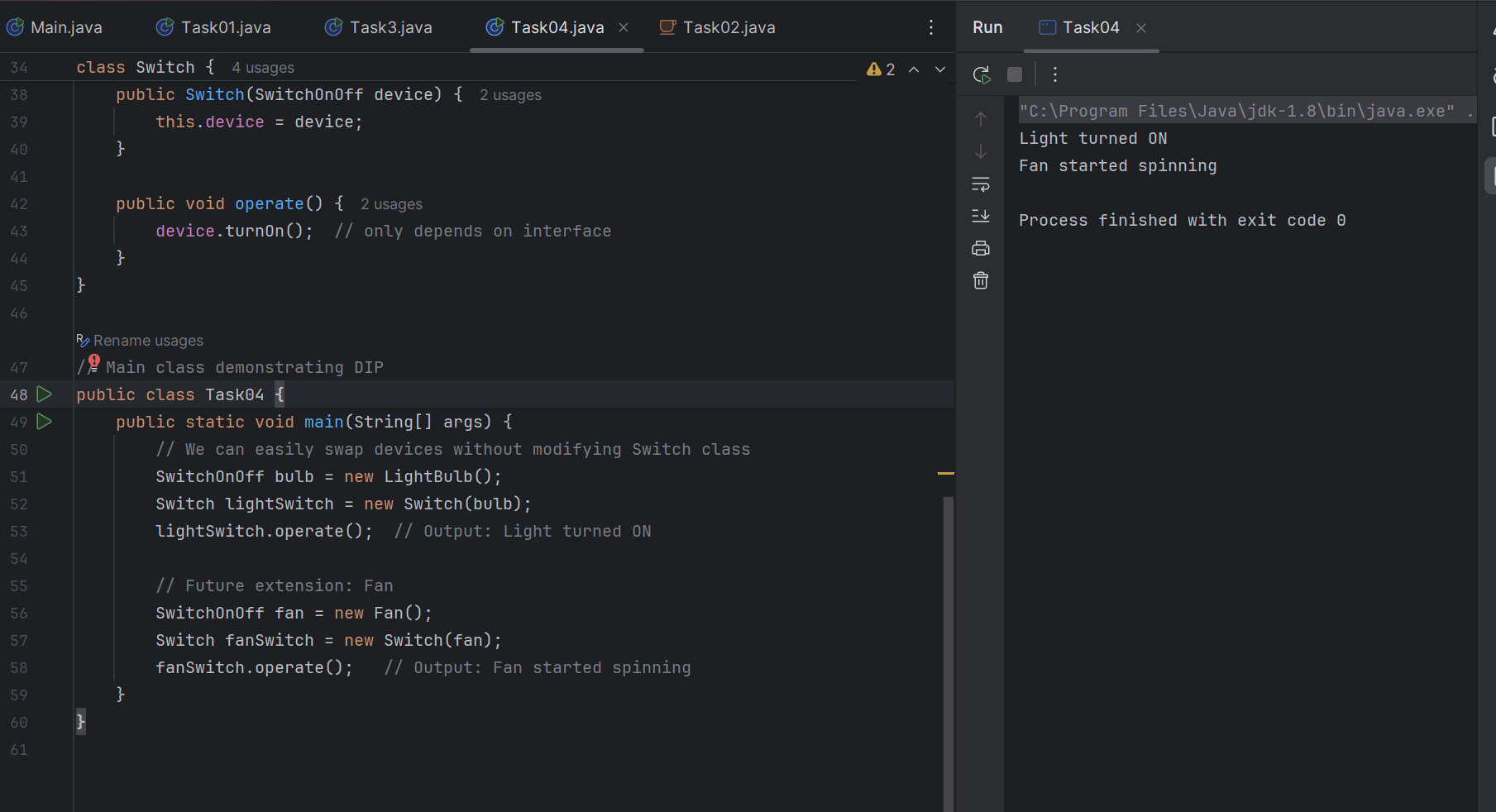
**Switch fanSwitch = new Switch(fan);**

**fanSwitch.operate(); // Output: Fan started spinning**

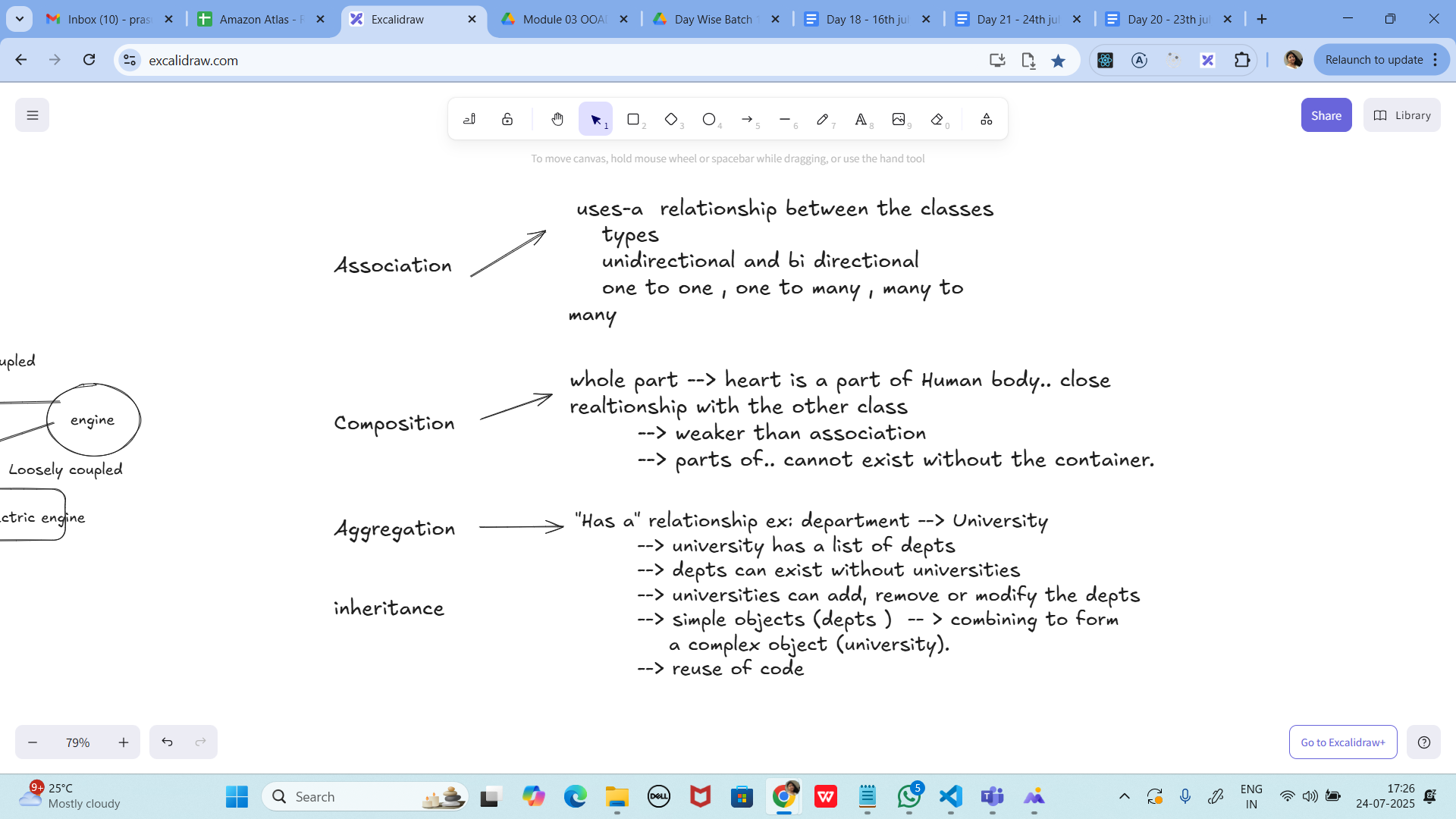
**}**

**}**

**Output:**



Association, Composition, Aggregation, Inheritance



Task 09:

Why should we choose Composition over Inheritance?

Loose coupling

Flexibility

Better encapsulation

## **1. Loose Coupling**

* Inheritance = tight coupling  
  + When a subclass inherits from a superclass, it depends heavily on the parent’s implementation.
  + Any change in the parent class may break the child.
* Composition = loose coupling  
  + In composition, a class contains another class rather than extending it.
  + The contained object can easily be replaced without affecting the main class.

*Example:*

A Car HAS-A Engine → You can swap the engine with another one (diesel, petrol, electric) without changing the Car class itself.

But if Car extends Engine, you’re locked to that hierarchy.

## **2. Flexibility**

* Inheritance is static  
  + Once you extend a class, the relationship is fixed at compile-time.
  + You cannot change behavior dynamically.
* Composition is dynamic  
  + You can compose different behaviors at runtime.
  + You can swap one object with another implementation.

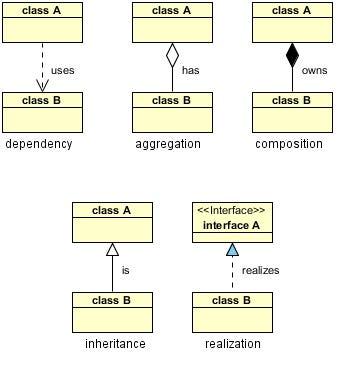
*Example:* A Printer class can have a PrintDriver (Inkjet or Laser). You can switch drivers at runtime, making it more flexible than a rigid inheritance hierarchy.

## **3. Better Encapsulation**

* Inheritance exposes parent details  
  + A child class can access protected or public fields/methods of the parent.
  + This can break encapsulation because subclass depends too much on the parent’s internals.
* Composition hides internal details  
  + The composed object is used through an interface, so only required functionality is exposed.
  + Internal changes of the composed class won’t affect the main class.

💡 *Example:* If a Car uses an Engine via an Engine interface, the Car doesn’t care about the exact implementation (PetrolEngine, DieselEngine, ElectricMotor). Encapsulation stays intact.

UML representation



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

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

Till 15.08…

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

Home Task:

Task 01

Refer the below doc in Docs to study

Doc 07 UML Class Diagrams

**Done**

Dependency: Based on the above image

Class A uses class B, but neither class A nor class B strictly owns each other. The relationship exists temporarily.

Task 02:

Aggregation: Based on the above image

Class A contains a reference to class B, but class B can exist independently of class A. This relationship is often described as “has-a.”

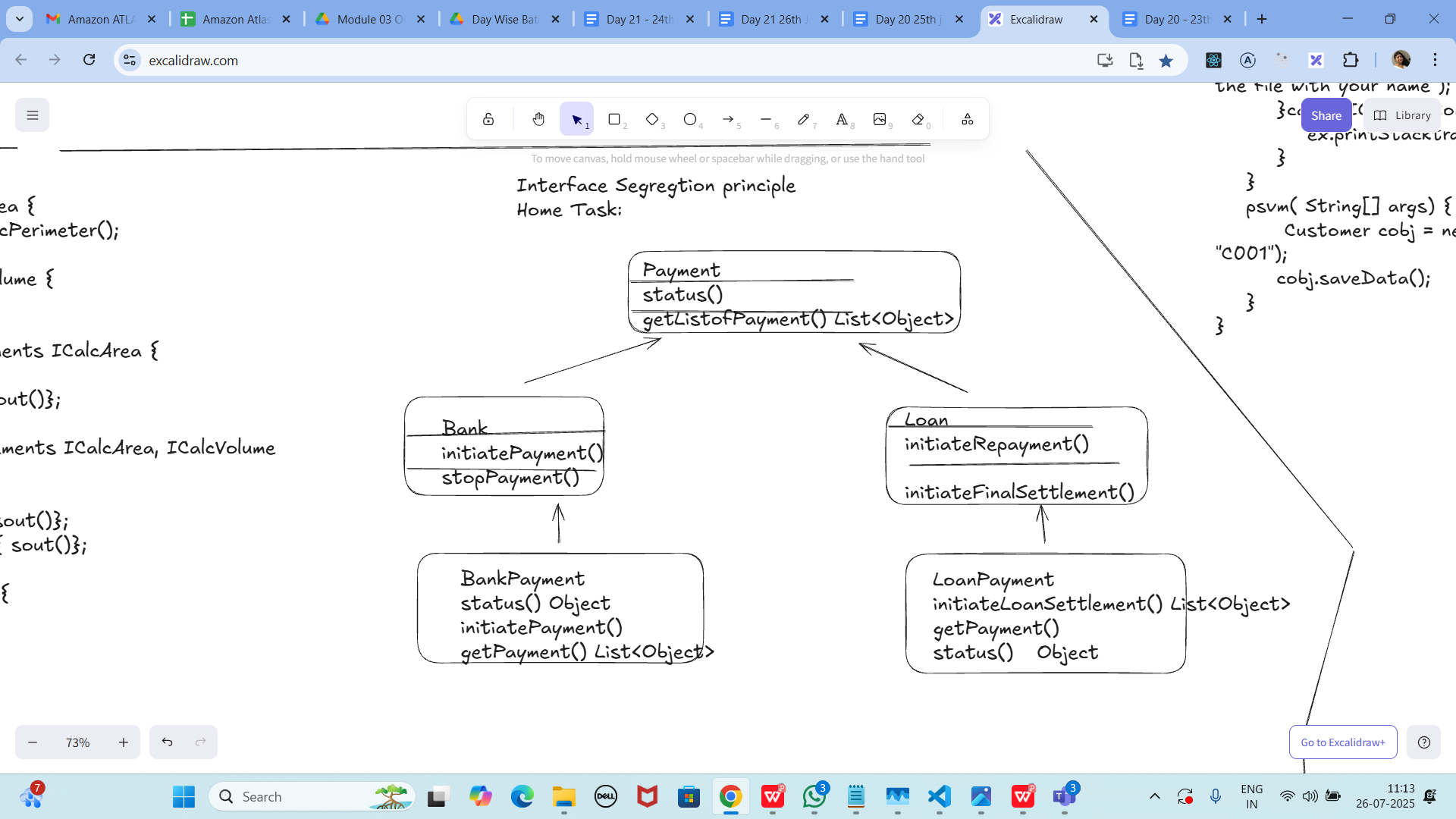
Task 03:

Composition : Based on the above image

Class A owns class B, meaning if class A is destroyed, class B will be too. This is a strong ownership relationship.

Task 04:

Based on the diagrammatic representation can you write a program.



**PROGRAM:**

**import java.util.List;**

**import java.util.ArrayList;**

**// Base interface for common payment behavior**

**interface Payment {**

**Object status();**

**List<Object> getListOfPayment();**

**}**

**// Interface specific to Bank payments**

**interface Bank {**

**void initiatePayment();**

**void stopPayment();**

**}**

**// Interface specific to Loan payments**

**interface Loan {**

**void initiateRepayment();**

**void initiateFinalSettlement();**

**}**

**// Implementation for Bank Payment**

**class BankPayment implements Payment, Bank {**

**@Override**

**public Object status() {**

**return "Bank Payment Status: SUCCESS";**

**}**

**@Override**

**public List<Object> getListOfPayment() {**

**List<Object> payments = new ArrayList<>();**

**payments.add("Bank Payment #1");**

**payments.add("Bank Payment #2");**

**return payments;**

**}**

**@Override**

**public void initiatePayment() {**

**System.*out*.println("Bank Payment initiated!");**

**}**

**@Override**

**public void stopPayment() {**

**System.*out*.println("Bank Payment stopped!");**

**}**

**}**

**// Implementation for Loan Payment**

**class LoanPayment implements Payment, Loan {**

**@Override**

**public Object status() {**

**return "Loan Payment Status: PENDING";**

**}**

**@Override**

**public List<Object> getListOfPayment() {**

**List<Object> loans = new ArrayList<>();**

**loans.add("Loan Payment #1");**

**loans.add("Loan Payment #2");**

**return loans;**

**}**

**@Override**

**public void initiateRepayment() {**

**System.*out*.println("Loan repayment initiated!");**

**}**

**@Override**

**public void initiateFinalSettlement() {**

**System.*out*.println("Loan final settlement initiated!");**

**}**

**}**

**// Test class**

**public class Task04 {**

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

**// Working with Bank Payment**

**BankPayment bankPayment = new BankPayment();**

**bankPayment.initiatePayment();**

**bankPayment.stopPayment();**

**System.*out*.println(bankPayment.status());**

**System.*out*.println("Bank Payments: " + bankPayment.getListOfPayment());**

**// Working with Loan Payment**

**LoanPayment loanPayment = new LoanPayment();**

**loanPayment.initiateRepayment();**

**loanPayment.initiateFinalSettlement();**

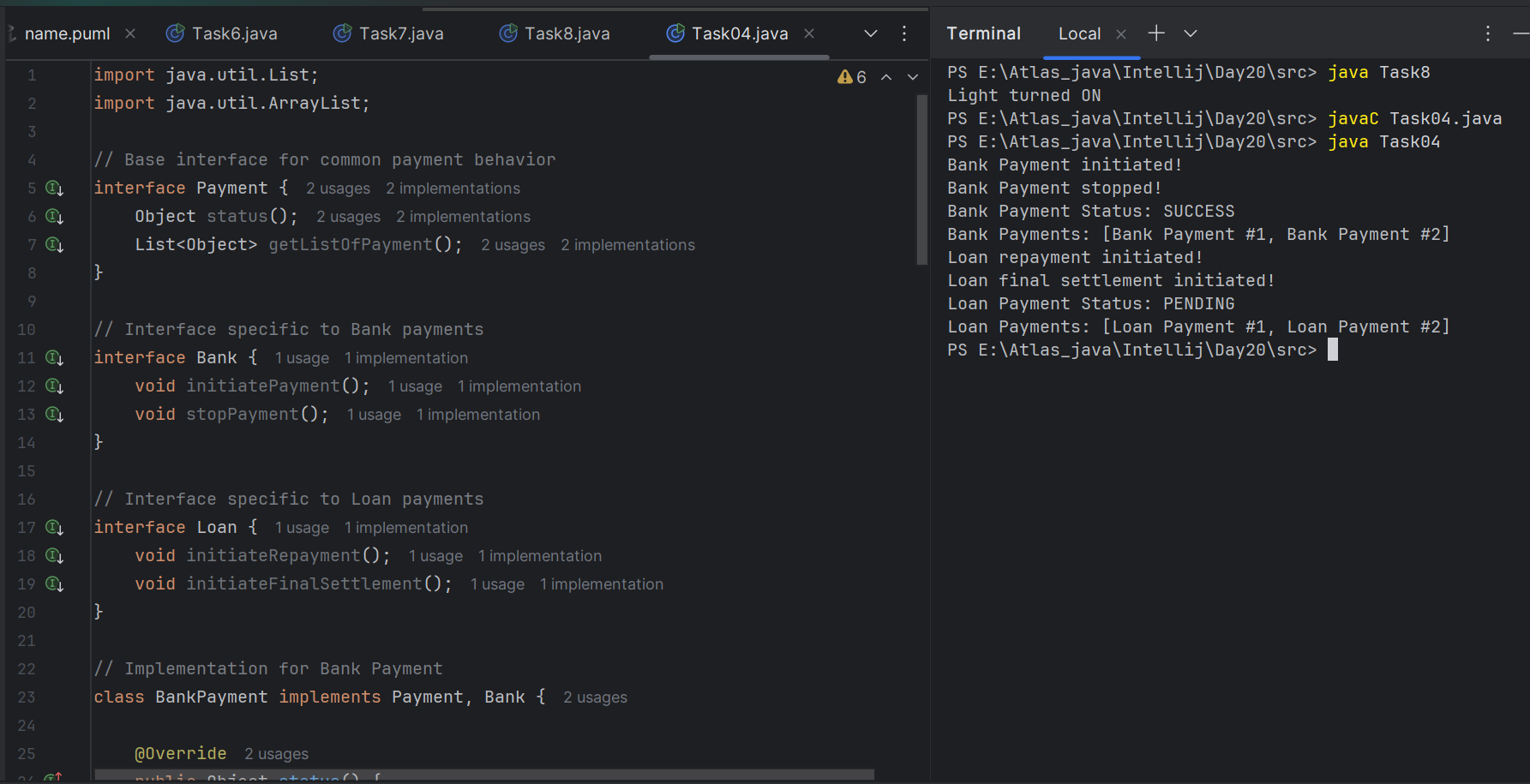
**System.*out*.println(loanPayment.status());**

**System.*out*.println("Loan Payments: " + loanPayment.getListOfPayment());**

**}**

**}**

**OUTPUT:**



Task 05:

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){

}

}



Task 06:

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");

}

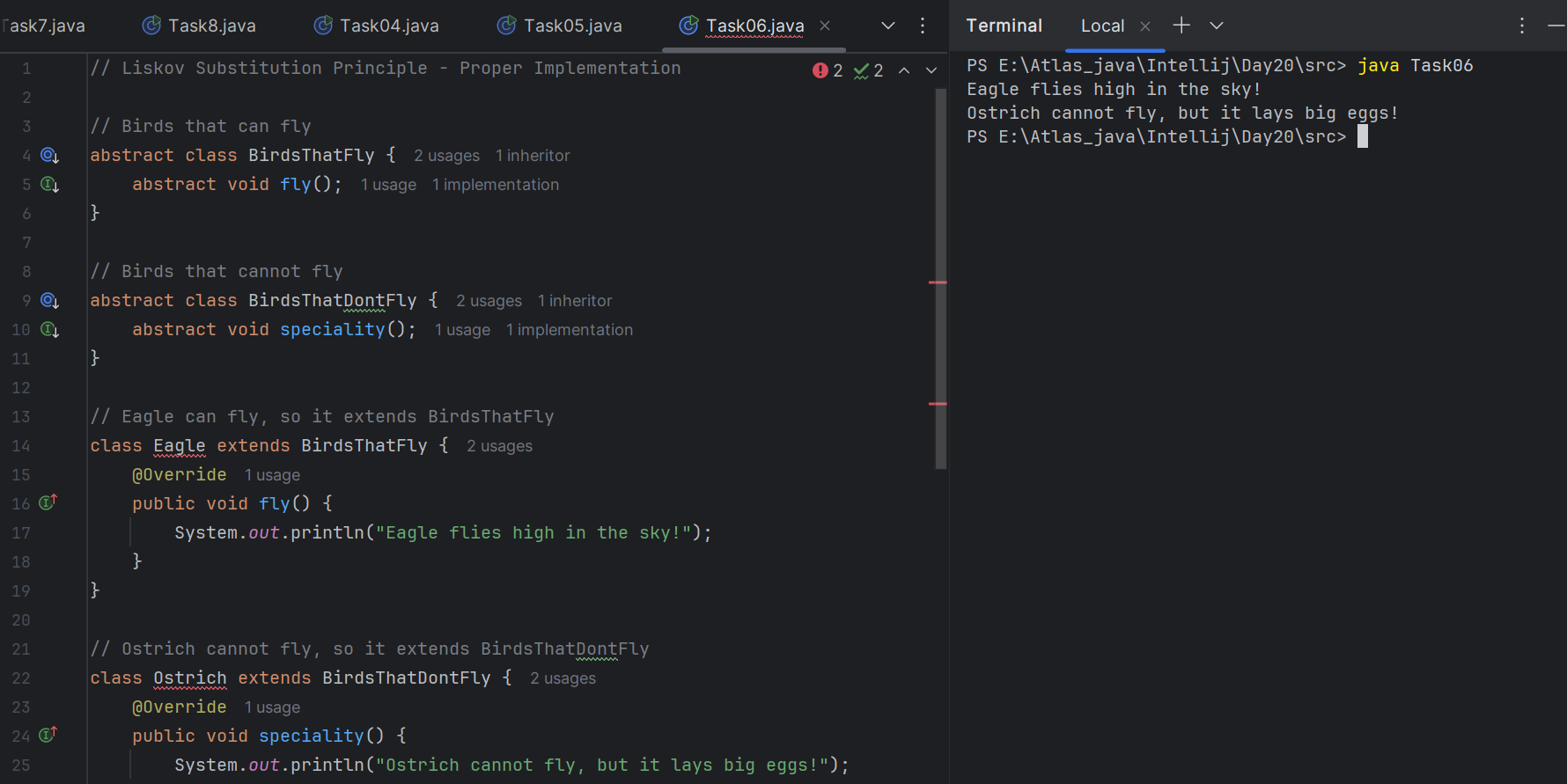
}

class Driverclass{

psvm(String[] args){

}

}



Task 07:

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()};

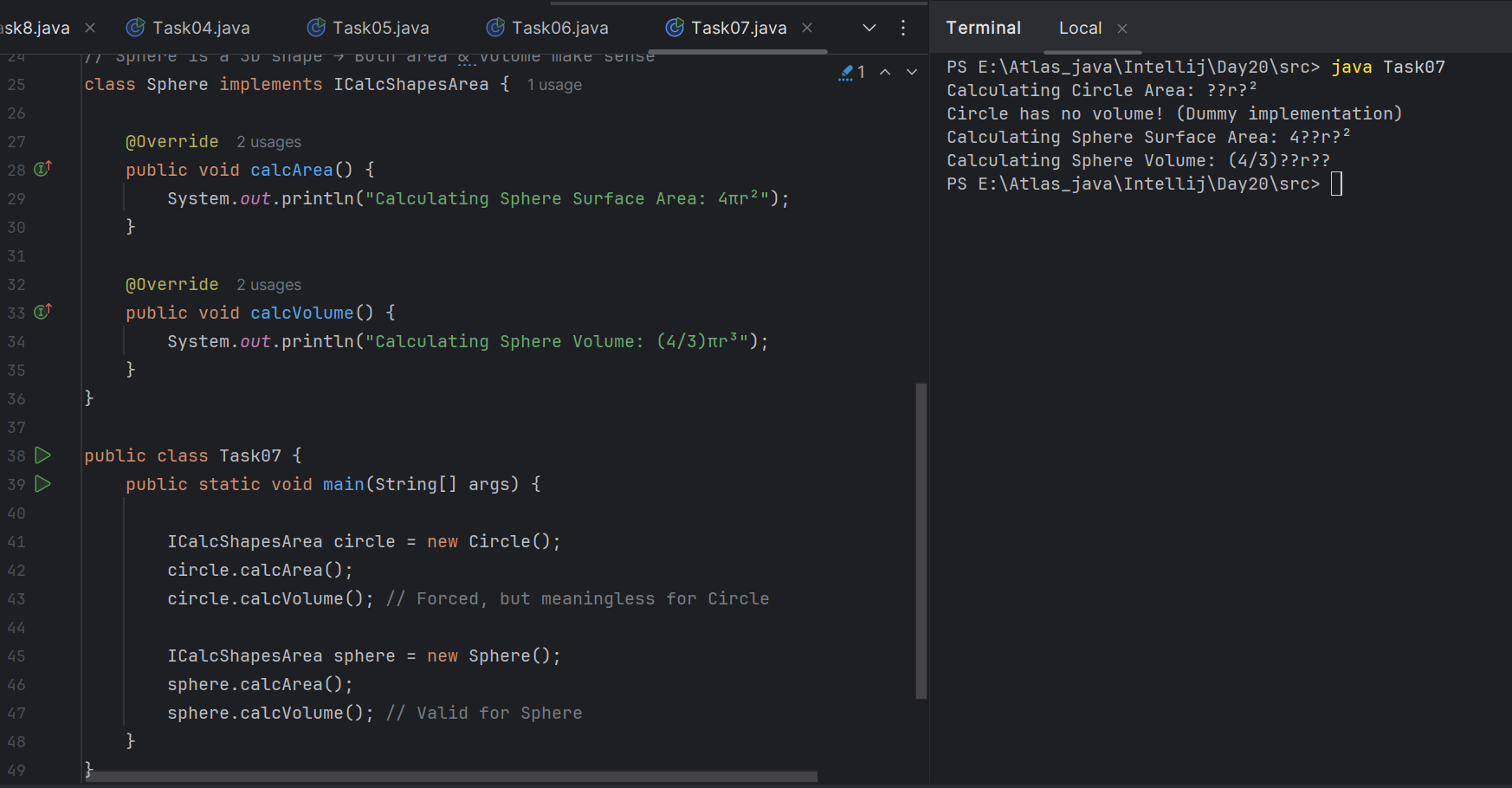
calcVolume() { sout()};

}

class Driverclass {

psvm( ,... ) {

}

}

Task 08:

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()};

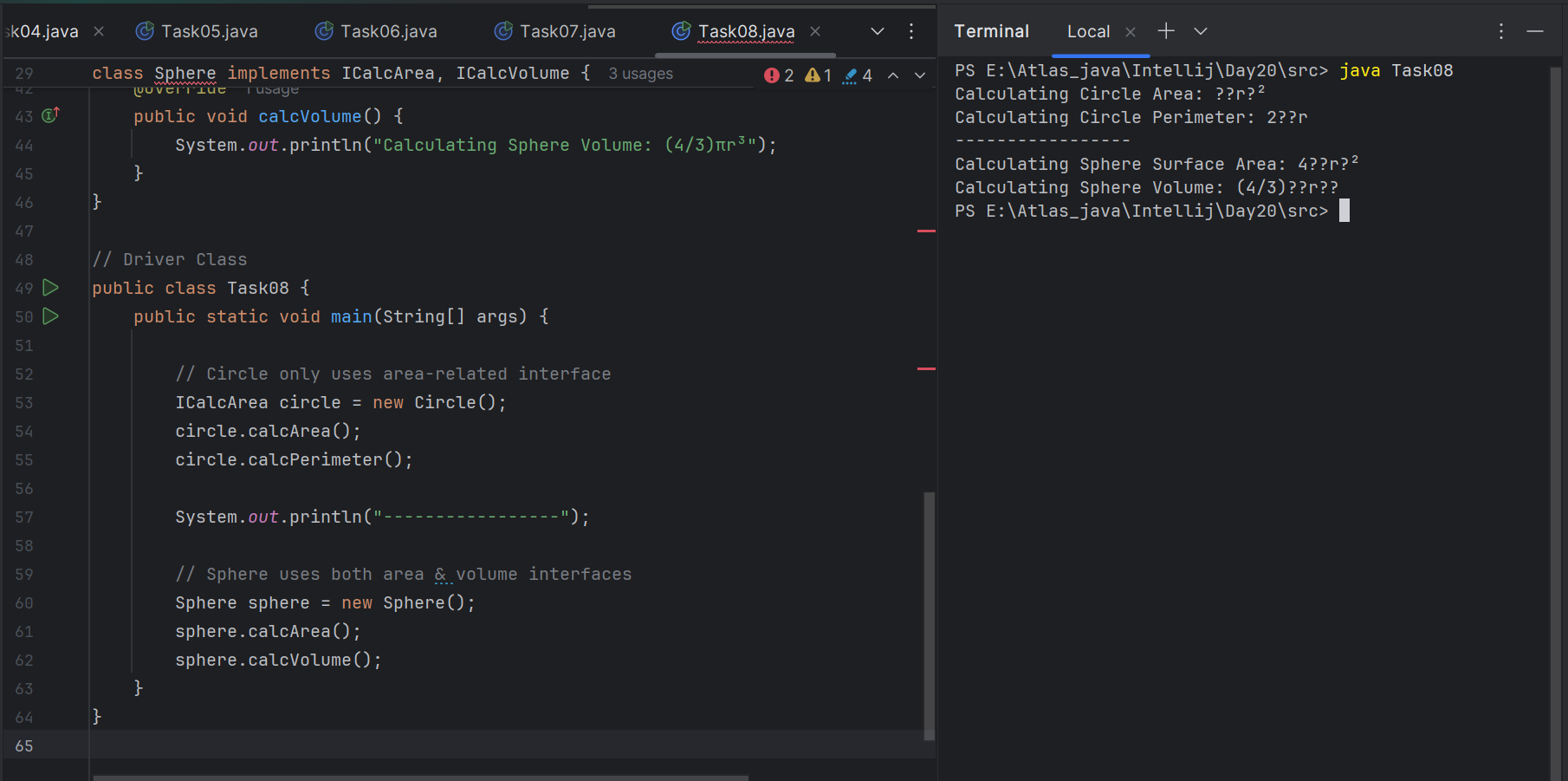
}

class Driverclass {

psvm( ,... ) {

}

}



Task 09:

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() {

}

}

**PROGRAM:**

**// Low-level module: Clothes**

**class Clothes {**

**void seeRating() {**

**System.*out*.println("Clothes rating is 4.5/5");**

**}**

**void viewSample() {**

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

**}**

**}**

**// High-level module: Cupboard (depends directly on Clothes)**

**class Cupboard {**

**Clothes cobj;**

**void addClothes(Clothes cobj) {**

**this.cobj = cobj;**

**System.*out*.println("Clothes added to cupboard.");**

**}**

**void customizeClothes() {**

**cobj.viewSample();**

**cobj.seeRating();**

**}**

**}**

**// New requirement: Add Books**

**class Books {**

**void seeRating() {**

**System.*out*.println("Book rating is 4.8/5");**

**}**

**void readSample() {**

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

**}**

**}**

**// Only ONE public class matching filename**

**public class Task09 {**

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

**Cupboard cupboard = new Cupboard();**

**// Initially supports only Clothes**

**Clothes clothes = new Clothes();**

**cupboard.addClothes(clothes);**

**cupboard.customizeClothes();**

**// If customer asks for Books → We need to MODIFY Cupboard → DIP Violation**

**Books book = new Books();**

**book.readSample();**

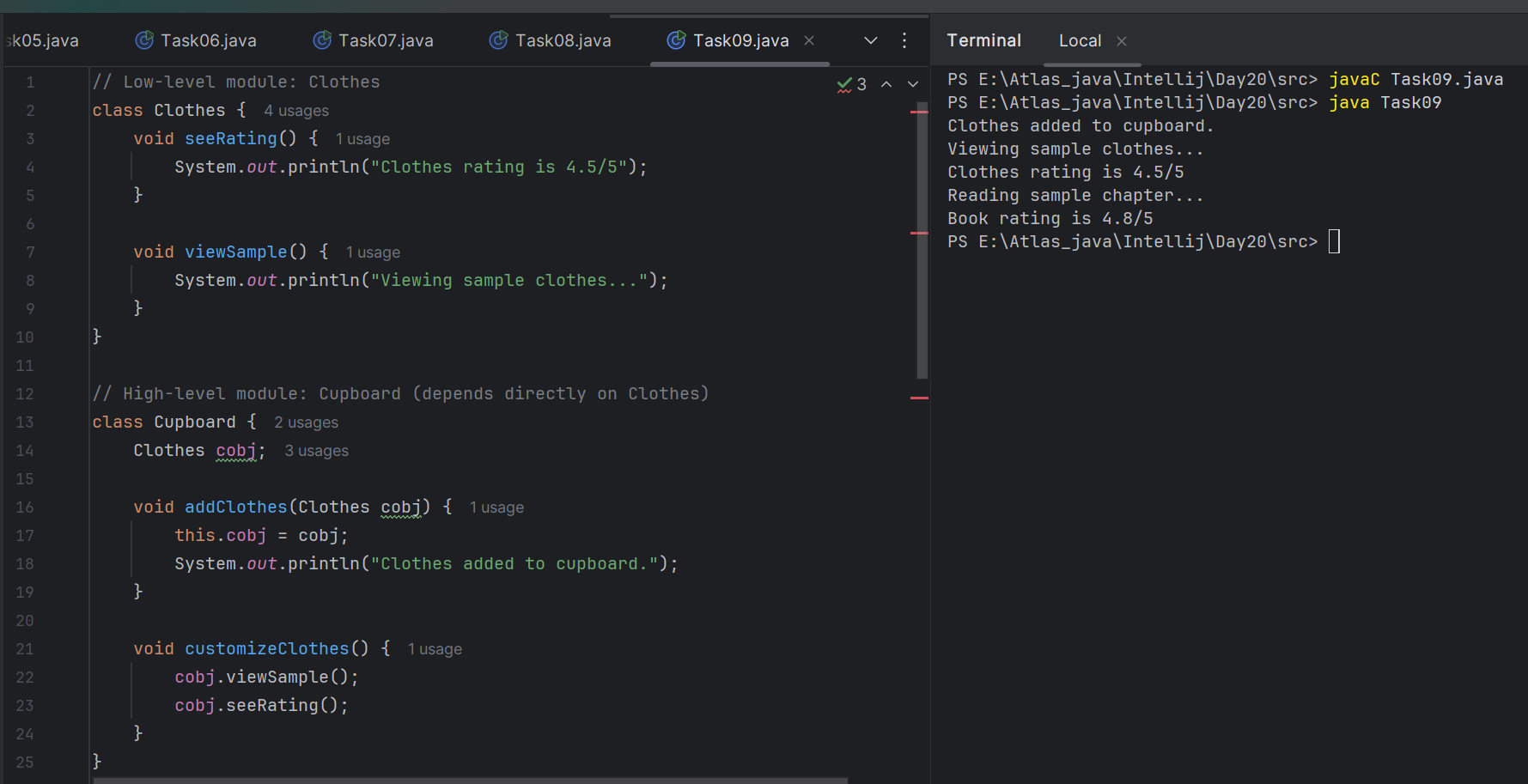
**book.seeRating();**

**// NOTE: Cupboard cannot handle Books without modifying its code.**

**}**

**}**

**OUTPUT:**

****

Task 10:

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() {

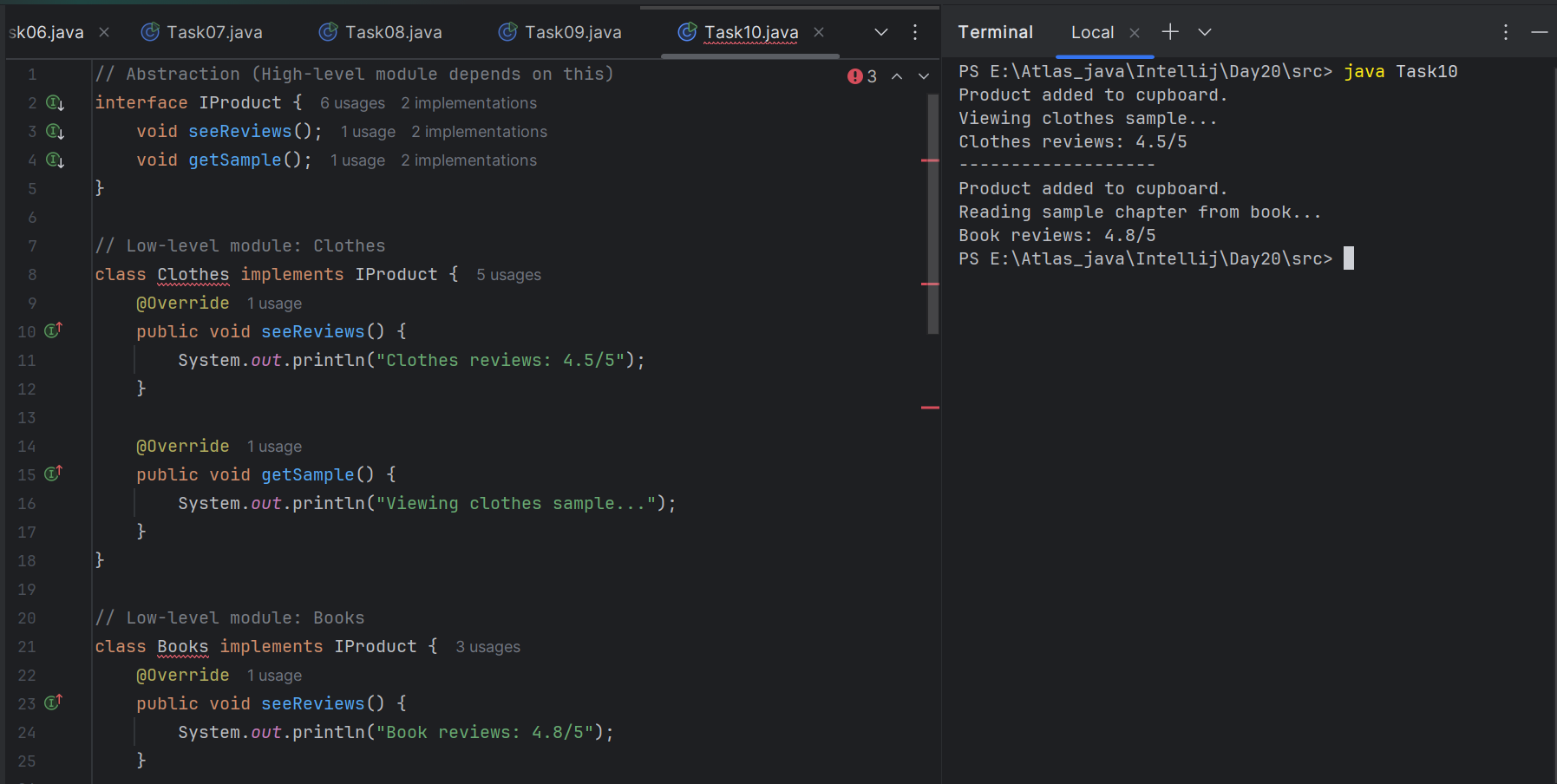
}

@Override

public void getSample() {

}

}



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

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

**Info box:**

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

<https://excalidraw.com/#json=_N42kljaUBYMEJbzBEG_Z,gBOJnMKaULoQxKebr7Hzsw>

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

## **6. Explain the different types of relationships in** [UML](https://www.geeksforgeeks.org/unified-modeling-language-uml-introduction/) **(association, aggregation, composition, inheritance)**

* [**Association**](https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/) **:** A relationship between two classes that indicates a connection or link between their objects.
* [**Aggregation**](https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/) **:** A special type of association that represents a "has-a" or "part-of" relationship, where one class is a part of another class, but they have separate lifetimes.
* [**Composition**](https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/) **:** A stronger form of aggregation, where the contained object's lifecycle is dependent on the container object.
* [**Inheritance**](https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/) **:** A relationship where one class (subclass) inherits the structure and behavior of another class (superclass), enabling code reuse and specialization.

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