**ID – hrajranj**

**Day 21 - 24th july 2025**

**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{

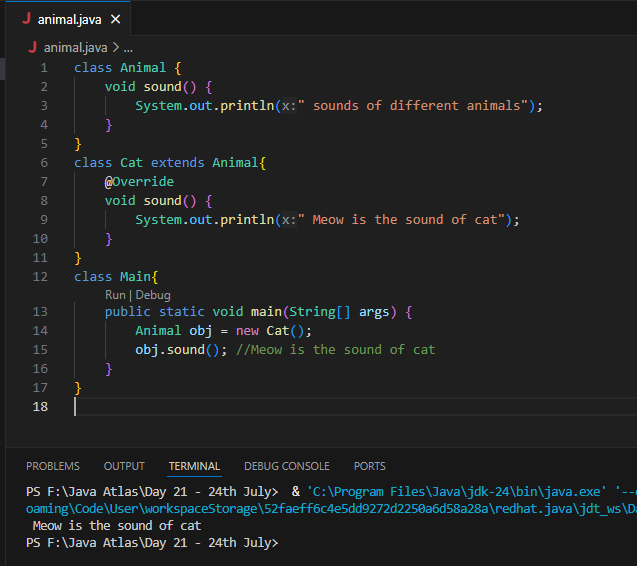
psvm(String[] args) {

Animal obj = new Cat();

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

}

}



**Task 02:**

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

void printList(List<?>  list) {

for(Object element: list) {

sout (element);

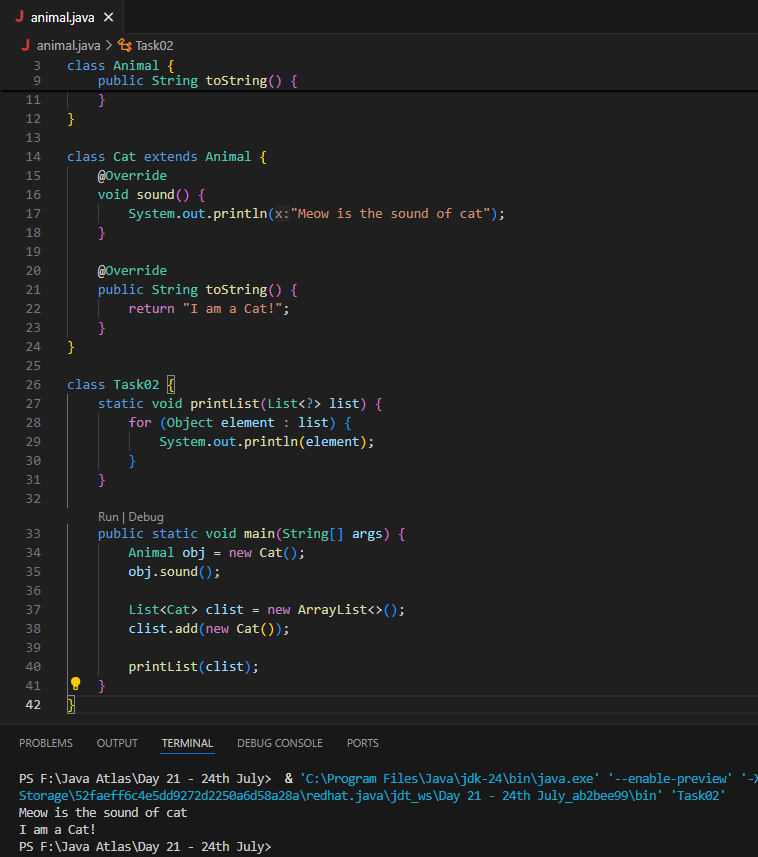
}

}

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

clist.add(new Cat());

printList(clist); //



**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); //meow

**Task 04:**

lower Bounded Wildcards

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

cats.add(new Cat());

}

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

addAcat(animals); //

**Task 05:**

// tight coupling

class Student {

public int roll\_no = 10;

//private int roll\_no = 10; // this code breaks...

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; // we are not following the rule of Student class roll no > 100..

//sobj.roll\_no = 110;

sout("the roll no of student is "+ sobj.roll\_no); // 110 // private - error //10

}

}

**Task 06:**

**// loose coupling**

**Solution:** Use **private fields with getters/setters** to control access.

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();

}

}

**Task 07:**

**//DIP violating**

The **Switch** class directly depends on **LightBulb.** If we need to add a **Fan**, we have to modify the **Switch class**, which violates **Dependency Inversion Principle (DIP).**

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();

}

}

**Task 09:**

**Why should we choose Composition over Inheritance?**

* Loose coupling
* Flexibility
* Better encapsulation

**Explain the different types of relationships in UML (association, aggregation, composition, inheritance):**

**Association:**

Association represents a general connection or link between two classes. It signifies that instances of one class can be related to instances of another class.

A general relationship between two independent classes. One class "knows about" the other class or "uses" the other class. But neither of the class owns the other.

Ex: A Student is associated with a Teacher. The student knows their teacher, but they can exist independently.

**Aggregation:**

Aggregation is a "part-of" relationship where one class (the whole or aggregate) contains or is composed of other classes (the parts).

It is like one class is a container, but parts can exist independently that is one class is a part of another class that can exist independently. It represents a "has-a" or "part-of" relationship.

Ex: A Department has many Professors. If the department closes, professors still exist (they can transfer).

**Composition:**

Composition is a stronger "part-of" relationship where the parts are entirely dependent on the whole for their existence.

It represents a strong whole-part relationship. The part cannot exist without the whole. If the whole is destroyed, the parts are too.

Ex: A House contains Rooms. If the house is demolished, rooms don’t exist separately.

**Inheritance:**

Inheritance is a relationship where a class (the subclass or derived class) inherits properties and behaviors from another class (the superclass or base class).

It represents a hierarchical relationship. One class (subclass) inherits behavior/attributes from another class (superclass).

Ex: A Dog is an Animal. Dog inherits common traits from Animal.