Day 21

Name: Aravind Kasanagottu

Id: MVSNARAV

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

}

}

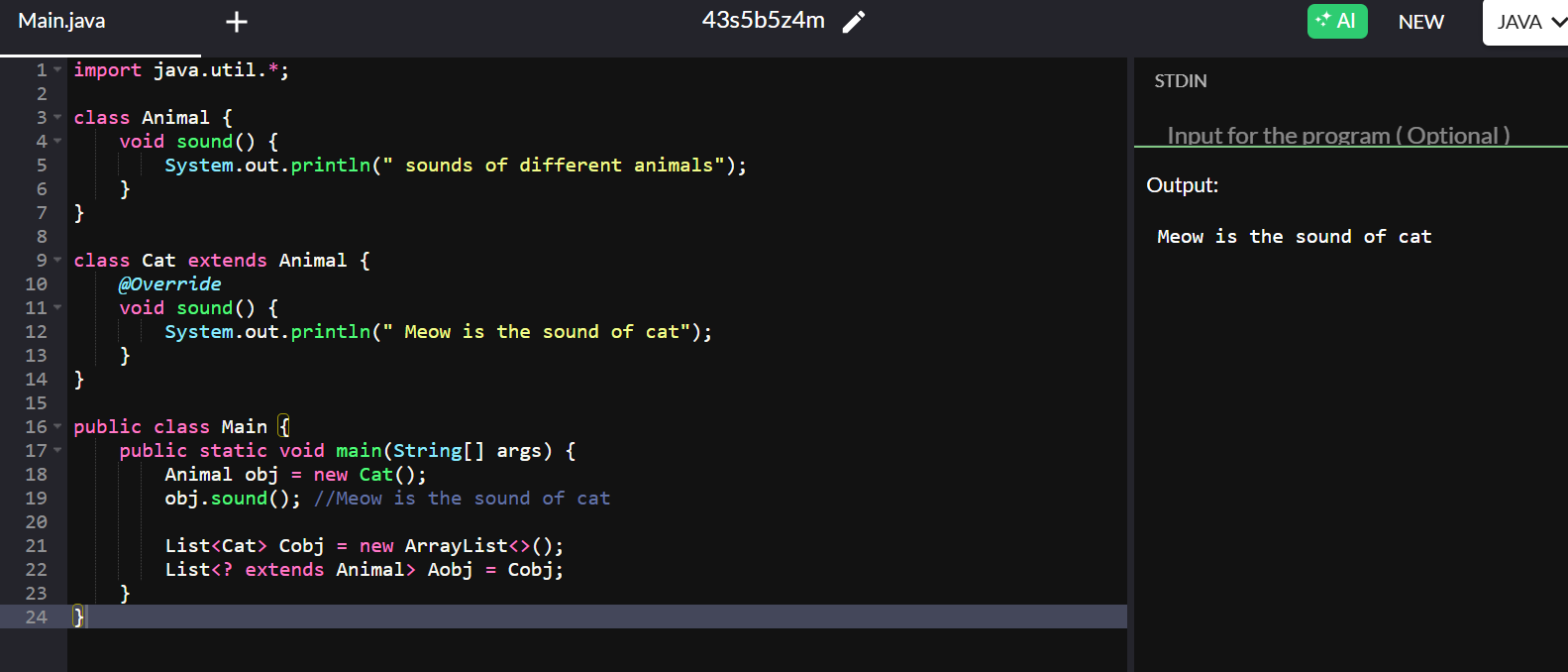
issue with Substitution  and Generics

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 ,



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);

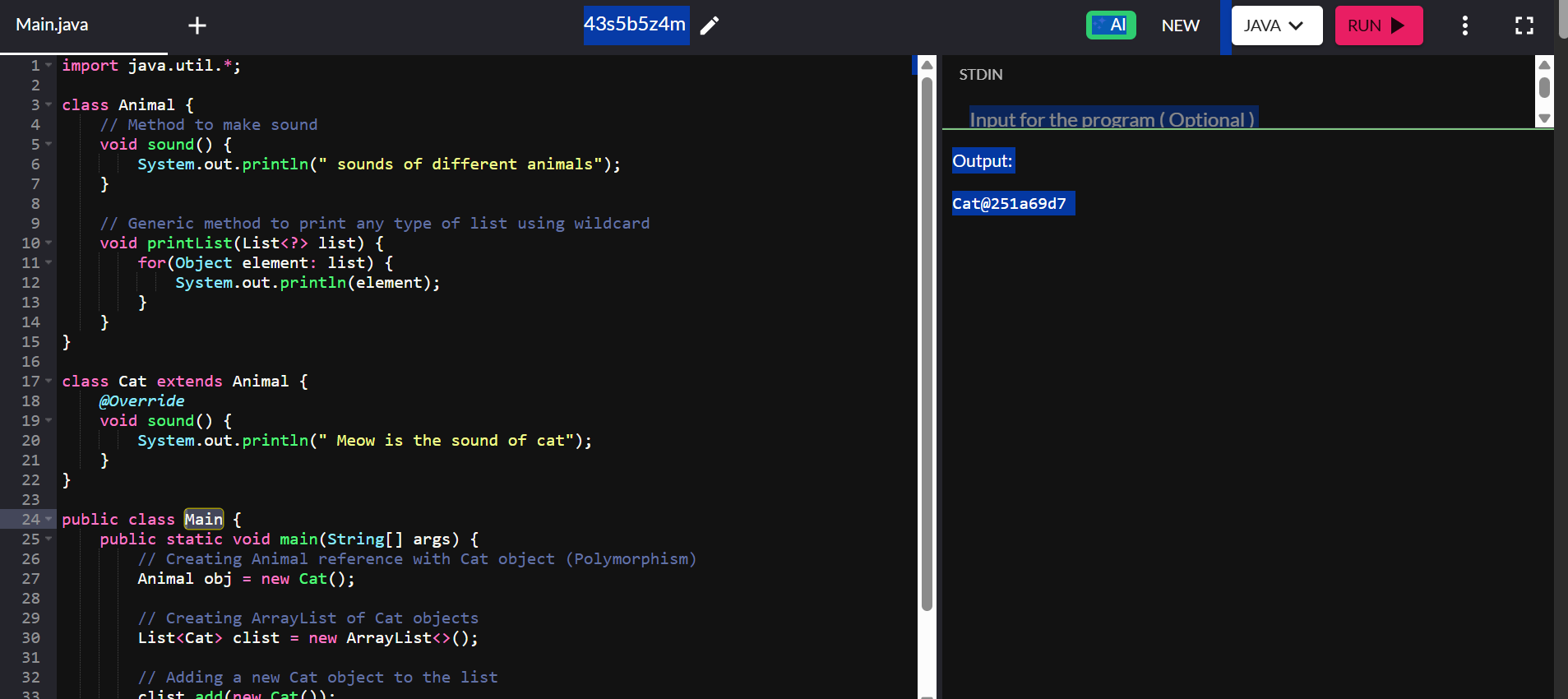
}

}

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();

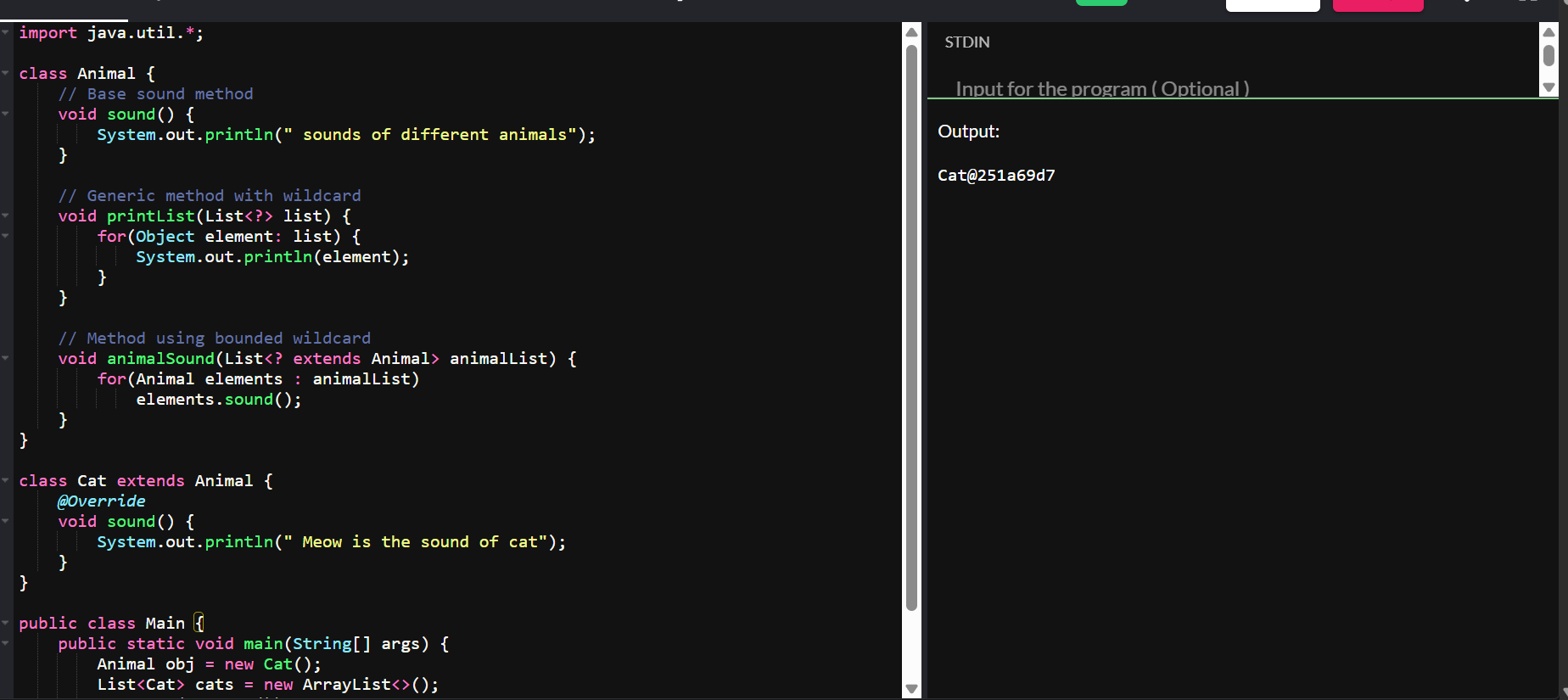
}

}

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

cats.add(new Cat());

animalSound(cats); //meow



\

Task 04:

lower Bounded Wildcards

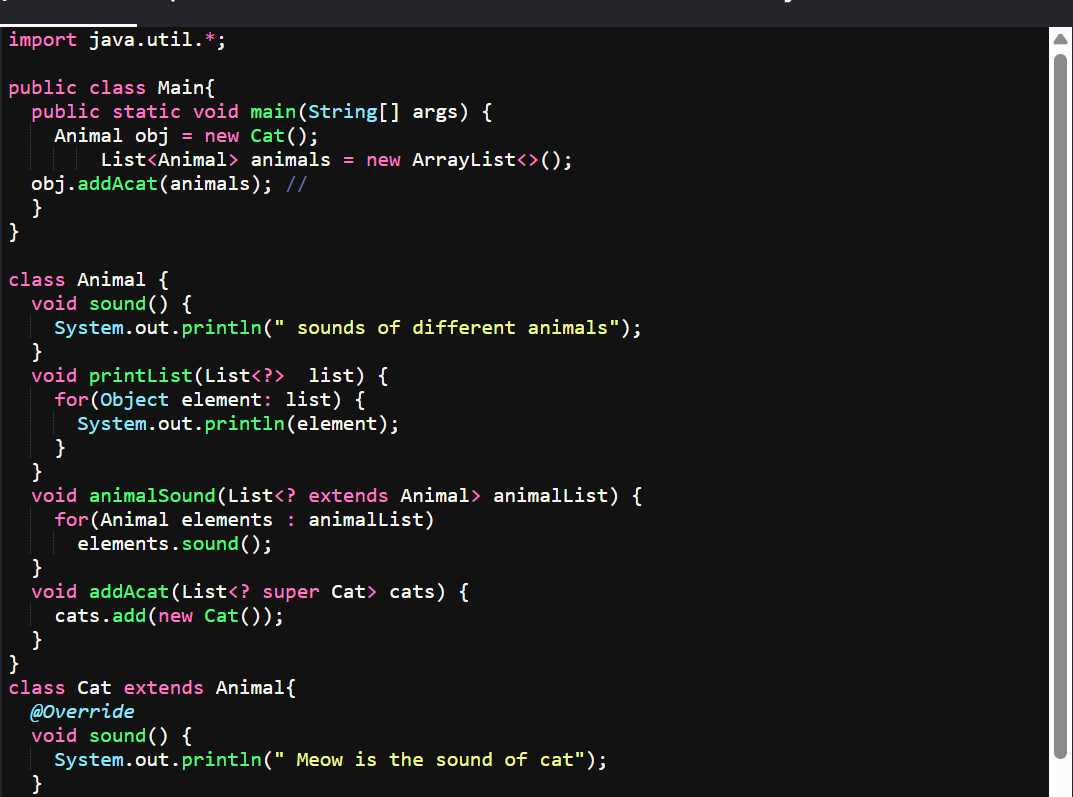
void addAcat(List<? super Cat> cats) {

cats.add(new Cat());

}

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

addAcat(animals); /



Task 05:

class Student {

public 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;

}

}

public class Tight\_coupling {

public static void main(String[] args) {

Student sobj = new Student();

sobj.roll\_no = 10;

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

}

}

Task 06:

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))

roll\_no = roll;

}

}

public class Loose\_coupling {

public static void main(String[] args) {

Student sobj = new Student();

sobj.setRoll(10);

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

}

}

Task 07:

class LightBulb {

void turnOn() {

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

}

void turnOff() {

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

}

}

class Switch {

LightBulb lbulbobj;

Switch(LightBulb lbulbobj) {

this.lbulbobj = lbulbobj;

}

void operate() {

lbulbobj.turnOn();

}

public static void main(String[] args) {

LightBulb lbulbobj = new LightBulb();

Switch switchObj = new Switch(lbulbobj);

switchObj.operate();

}

}

Task 08:

interface SwitchOnOff {

void turnOn();

void turnOff();

}

class LightBulb implements SwitchOnOff {

@Override

public void turnOn() {

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

}

@Override

public void turnOff() {

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

}

}

class Fan implements SwitchOnOff {

@Override

public void turnOn() {

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

}

@Override

public void turnOff() {

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

}

}

class Switch {

private SwitchOnOff device;

public Switch(SwitchOnOff device) {

this.device = device;

}

public void operate() {

device.turnOn();

}

}

public class DIP {

public static void main(String[] args) {

SwitchOnOff lightBulb = new LightBulb();

Switch lightSwitch = new Switch(lightBulb);

lightSwitch.operate();

SwitchOnOff fan = new Fan();

Switch fanSwitch = new Switch(fan);

fanSwitch.operate();

}

}

Task 9:

Why should we choose Composition over Inheritance?

1. **Loose Coupling**
   * Inheritance: Creates tight parent-child dependency
   * Composition: Objects remain independent, interact via interfaces
   * Result: Easier to modify and maintain code
2. **Flexibility**
   * Inheritance: Fixed at compile time
   * Composition: Can change behavior at runtime
   * Result: More adaptable to changing requirements
3. **Better Encapsulation**
   * Inheritance: Child classes can access parent's internals
   * Composition: Objects interact only through public interfaces
   * Result: Better information hiding and security