**day21\_107856406\_dsdipt\_sudipto\_24july2025**

**Employee Code:** 107856406

**Login ID:** dsdipt

**Email :** dsdipt@amazon.com

**Name:** Sudipto Das

**Date:** 24 July 2025 (Day 21)

### ***Task 01: Understanding Substitution Principle and Wildcards in Java Generics***

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");

}

}

class Main {

public static void main(String[] args) {

Animal obj = new Cat();

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

}

}

Problem with Generics and Substitution

In normal class inheritance, Cat is a subtype of Animal. But with generics:

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

List<Animal> animals = cats; // Compile error

**Why?** Because if this was allowed, you could add a Dog to a List<Cat>, which breaks type safety.  
 Java generics are **invariant**: List<Cat> is **not** a subtype of List<Animal>.

### 

### ***Task 02: Unbounded Wildcards***

Use ? when the **type does not matter** for reading elements.

void printList(List<?> list) {

for (Object element : list) {

System.out.println(element);

}

}

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

clist.add(new Cat());

printList(clist); // Works

### ***Task 03: Upper Bounded Wildcards (? extends Type)***

Use ? extends when you want to **read** elements as a base type but don't need to add new elements.

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

for (Animal element : animalList) {

element.sound();

}

}

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

cats.add(new Cat());

animalSound(cats); // Output: Meow is the sound of cat

### ***Task 04: Lower Bounded Wildcards (? super Type)***

Use ? super when you want to **add** elements of a given type or its subtypes.

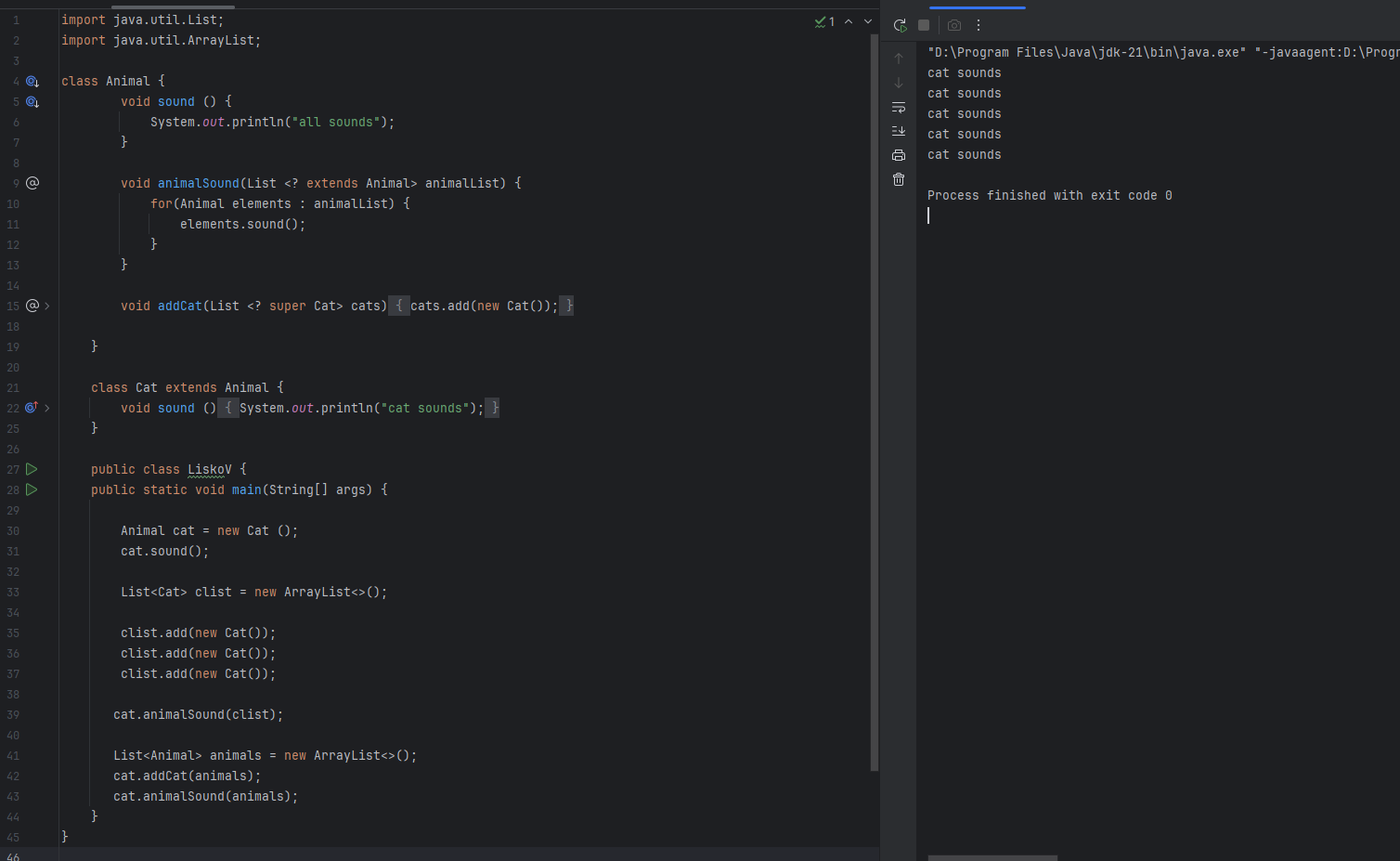
void addACat(List<? super Cat> cats) {

cats.add(new Cat()); // Safe to add Cat or subtypes

}

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

addACat(animals);



### ***Task 05: Tight Coupling***

**Problem:** Directly accessing roll\_no breaks encapsulation and the rules set by the Student class.

class Student {

public int roll\_no = 10; // Tight coupling - accessible directly

public int getRoll() {

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

return roll\_no;

}

public void setRoll(int roll) {

if (!(roll > 100))

roll\_no = roll;

}

}

class TightCoupling {

public static void main(String[] args) {

Student sobj = new Student();

sobj.roll\_no = 10; // Direct access - violates Student's rules

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

}

}

### ***Task 06: Loose Coupling***

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

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;

}

}

class LooseCoupling {

public static void main(String[] args) {

Student sobj = new Student();

sobj.setRoll(10);

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

}

}

Advantages of Loose Coupling:

* Better maintainability
* Easier testing
* Fewer error propagation issues
* Higher reusability

### ***Task 07: DIP Violation***

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() {

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

}

void turnOff() {

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

}

}

class Switch { // Directly depends on LightBulb (DIP violation)

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: DIP Implementation***

We fix the above by **depending on abstractions (interfaces)** instead of concrete classes.

interface SwitchOnOff {

void turnOn();

void turnOff();

}

class LightBulb implements SwitchOnOff {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

class Switch {

private SwitchOnOff device;

Switch(SwitchOnOff device) {

this.device = device;

}

void operate() {

device.turnOn();

}

}

class DIP {

public static void main(String[] args) {

SwitchOnOff lbulbobj = new LightBulb();

Switch lightSwitch = new Switch(lbulbobj);

lightSwitch.operate();

}

}

### ***Task 09: Why Choose Composition over Inheritance?***

**Inheritance: "Is-A" Relationship**

* Inheritance means a subclass **is a type of** its superclass.

**Example:**  
  
 class Animal { void sound() {} }

class Dog extends Animal { void sound() { System.out.println("Bark"); } }

* Here, Dog **is an Animal**, which is fine.

**But inheritance is rigid:**

* If the parent changes, all child classes might break.
* You get **tight coupling** between child and parent.
* You can only inherit **one parent class** (Java doesn't allow multiple inheritance with classes).

**Composition: "Has-A" Relationship**

* Composition means a class **has** another class as a field and **delegates behavior** to it.

**Example:**  
  
 class Engine {

void start() { System.out.println("Engine starts..."); }

}

class Car {

private Engine engine = new Engine(); // Car HAS-A Engine

void startCar() { engine.start(); }

}

**With composition:**

* You can **easily swap components** (e.g., replace Engine with ElectricEngine).
* It’s more **flexible and reusable**.
* No fragile hierarchy or unexpected method overrides.

**Why Prefer Composition?**

* **Fewer side effects:** Changes in one class don’t break others.
* **Reusability:** You can reuse behavior without forcing "is-a" relationships.
* **Flexibility:** You can combine different classes dynamically (e.g., at runtime).
* **Avoids deep hierarchies:** Inheritance often leads to complex, hard-to-maintain trees.

**Quick Example: Inheritance vs Composition**

Using Inheritance

class Bird {

void fly() { System.out.println("I can fly"); }

}

class Penguin extends Bird {

// Oops! Penguins don't fly, but they inherit fly() anyway. 🙃

}

This design fails because **Penguin is not a bird that flies**.

Using Composition

interface Flyable { void fly(); }

class CanFly implements Flyable { public void fly() { System.out.println("I can fly"); } }

class CannotFly implements Flyable { public void fly() { System.out.println("I can't fly"); } }

class Bird {

private Flyable flyingType;

Bird(Flyable flyingType) { this.flyingType = flyingType; }

void tryFlying() { flyingType.fly(); }

}

class Penguin extends Bird {

Penguin() { super(new CannotFly()); }

}

Now **Penguin** behaves correctly without inheriting the wrong behavior.

**Use inheritance when:**

* The relationship is naturally "is-a".
* Behavior is stable and won’t change unexpectedly.

**Use composition when:**

* You want **flexible, modular, and reusable code**.
* You want to avoid tight coupling and rigid hierarchies.