

**Core OOP Principles - Inheritance - Practice Problems - Any Two**

�� **PRACTICE PROBLEM 1: Single Inheritance with extends and super**

**Understanding basic inheritance, constructor chaining, and super keyword usage**

// File: Vehicle.java

public class Vehicle {

// TODO: Create protected fields for inheritance:

// - brand (String) - accessible to subclasses

// - model (String) - accessible to subclasses

// - year (int) - accessible to subclasses

// - engineType (String) - accessible to subclasses

// TODO: Create private fields that require getter/setter access: // - registrationNumber (String) - only through methods // - isRunning (boolean) - internal state

// TODO: Create default constructor that:

// - Sets default values for all fields

// - Prints "Vehicle default constructor called"

// TODO: Create parameterized constructor that:

// - Takes brand, model, year, engineType parameters

// - Initializes all fields

// - Prints "Vehicle parameterized constructor called" // - Generates random registration number

// TODO: Create methods for basic vehicle operations:

// - start() - sets isRunning to true, prints "Vehicle started" // - stop() - sets isRunning to false, prints "Vehicle stopped" // - getVehicleInfo() - returns formatted string with all vehicle details

1



// - displaySpecs() - prints technical specifications

// TODO: Create getter/setter methods for private fields: // - getRegistrationNumber() / setRegistrationNumber()

// - isRunning() - no setter for this (controlled through start/stop) }

// TODO: Create Car class that extends Vehicle:

public class Car extends Vehicle {

// TODO: Add car-specific fields:

// - numberOfDoors (int)

// - fuelType (String)

// - transmissionType (String)

// TODO: Create default constructor that:

// - Calls super() explicitly

// - Sets car-specific default values

// - Prints "Car default constructor called"

// TODO: Create parameterized constructor that:

// - Takes all Vehicle parameters plus car-specific parameters // - Calls super(brand, model, year, engineType) explicitly // - Initializes car-specific fields

// - Prints "Car parameterized constructor called"

// TODO: Override parent methods where appropriate:

// - Override start() to include car-specific startup sequence // - Call super.start() first, then add car-specific operations // - Override displaySpecs() to show both vehicle and car specifications

// TODO: Add car-specific methods:

// - openTrunk() - prints "Trunk opened"

// - playRadio() - prints "Radio playing music"

public static void main(String[] args) {

// TODO: Test constructor chaining:

// 1. Create Car using default constructor (observe constructor call order)

// 2. Create Car using parameterized constructor

2



// TODO: Test inheritance of fields and methods:

// - Access protected fields from parent

// - Call inherited methods

// - Call overridden methods

// TODO: Test super keyword usage:

// - Show how overridden methods can call parent implementation // - Demonstrate constructor chaining with super()

// TODO: Test method resolution:

// - Call methods that exist only in Car

// - Call methods that exist in both Vehicle and Car

// - Show polymorphic behavior

}

}

PPROGRAM:

// File: Vehicle.java

import java.util.UUID;

public class Vehicle {

protected String brand;

protected String model;

protected int year;

protected String engineType;

private String registrationNumber;

private boolean isRunning;

public Vehicle() {

this.brand = "Unknown";

this.model = "Unknown";

this.year = 0;

this.engineType = "Unknown";

this.registrationNumber = generateRegistrationNumber();

this.isRunning = false;

System.out.println("Vehicle default constructor called");

}

public Vehicle(String brand, String model, int year, String engineType) {

this.brand = brand;

this.model = model;

this.year = year;

this.engineType = engineType;

this.registrationNumber = generateRegistrationNumber();

this.isRunning = false;

System.out.println("Vehicle parameterized constructor called");

}

private String generateRegistrationNumber() {

return "REG-" + UUID.randomUUID().toString().substring(0, 8).toUpperCase();

}

public void start() {

isRunning = true;

System.out.println("Vehicle started");

}

public void stop() {

isRunning = false;

System.out.println("Vehicle stopped");

}

public String getVehicleInfo() {

return String.format("Brand: %s, Model: %s, Year: %d, Engine: %s, RegNo: %s, Running: %b",

brand, model, year, engineType, registrationNumber, isRunning);

}

public void displaySpecs() {

System.out.println("Vehicle Specifications:");

System.out.println("Brand: " + brand);

System.out.println("Model: " + model);

System.out.println("Year: " + year);

System.out.println("Engine Type: " + engineType);

}

public String getRegistrationNumber() {

return registrationNumber;

}

public void setRegistrationNumber(String registrationNumber) {

this.registrationNumber = registrationNumber;

}

public boolean isRunning() {

return isRunning;

}

}

// File: Car.java

public class Car extends Vehicle {

private int numberOfDoors;

private String fuelType;

private String transmissionType;

public Car() {

super();

this.numberOfDoors = 4;

this.fuelType = "Petrol";

this.transmissionType = "Manual";

System.out.println("Car default constructor called");

}

public Car(String brand, String model, int year, String engineType,

int numberOfDoors, String fuelType, String transmissionType) {

super(brand, model, year, engineType);

this.numberOfDoors = numberOfDoors;

this.fuelType = fuelType;

this.transmissionType = transmissionType;

System.out.println("Car parameterized constructor called");

}

@Override

public void start() {

super.start();

System.out.println("Car-specific startup checks complete (doors locked, seatbelts fastened)");

}

@Override

public void displaySpecs() {

super.displaySpecs();

System.out.println("Car Specifications:");

System.out.println("Number of Doors: " + numberOfDoors);

System.out.println("Fuel Type: " + fuelType);

System.out.println("Transmission: " + transmissionType);

}

public void openTrunk() {

System.out.println("Trunk opened");

}

public void playRadio() {

System.out.println("Radio playing music");

}

public static void main(String[] args) {

System.out.println("=== Test 1: Default Constructor ===");

Car car1 = new Car();

car1.displaySpecs();

car1.start();

car1.stop();

car1.openTrunk();

car1.playRadio();

System.out.println(car1.getVehicleInfo());

System.out.println("\n=== Test 2: Parameterized Constructor ===");

Car car2 = new Car("Toyota", "Camry", 2022, "Hybrid", 4, "Hybrid", "Automatic");

car2.displaySpecs();

car2.start();

car2.stop();

car2.openTrunk();

car2.playRadio();

System.out.println(car2.getVehicleInfo());

System.out.println("\n=== Test 3: Polymorphism ===");

Vehicle v = new Car("Honda", "Civic", 2023, "Petrol", 4, "Petrol", "Manual");

v.displaySpecs();

v.start();

System.out.println(v.getVehicleInfo());

}

}

OUTPUT:

=== Test 1: Default Constructor ===

Vehicle default constructor called

Car default constructor called

Vehicle Specifications:

Brand: Unknown

Model: Unknown

Year: 0

Engine Type: Unknown

Car Specifications:

Number of Doors: 4

Fuel Type: Petrol

Transmission: Manual

Vehicle started

Car-specific startup checks complete (doors locked, seatbelts fastened)

Vehicle stopped

Trunk opened

Radio playing music

Brand: Unknown, Model: Unknown, Year: 0, Engine: Unknown, RegNo: REG-8F2A9C1B, Running: false

=== Test 2: Parameterized Constructor ===

Vehicle parameterized constructor called

Car parameterized constructor called

Vehicle Specifications:

Brand: Toyota

Model: Camry

Year: 2022

Engine Type: Hybrid

Car Specifications:

Number of Doors: 4

Fuel Type: Hybrid

Transmission: Automatic

Vehicle started

Car-specific startup checks complete (doors locked, seatbelts fastened)

Vehicle stopped

Trunk opened

Radio playing music

Brand: Toyota, Model: Camry, Year: 2022, Engine: Hybrid, RegNo: REG-5E7B12A4, Running: false

=== Test 3: Polymorphism ===

Vehicle parameterized constructor called

Car parameterized constructor called

Vehicle Specifications:

Brand: Honda

Model: Civic

Year: 2023

Engine Type: Petrol

Car Specifications:

Number of Doors: 4

Fuel Type: Petrol

Transmission: Manual

Vehicle started

Car-specific startup checks complete (doors locked, seatbelts fastened)

Brand: Honda, Model: Civic, Year: 2023, Engine: Petrol, RegNo: REG-AB92C71F, Running: true

�� **PRACTICE PROBLEM 2: Multilevel Inheritance Chain Building deep inheritance hierarchies with proper constructor chaining** // File: MultilevelInheritanceDemo.java

// TODO: Create base class Animal:

public class Animal {

// TODO: Create protected fields:

// - species (String)

// - habitat (String)

// - lifespan (int)

// - isWildlife (boolean)

// TODO: Create constructor that:

// - Takes all parameters

// - Prints "Animal constructor: Creating [species]"

// TODO: Create methods:

// - eat() - prints "Animal is eating"

// - sleep() - prints "Animal is sleeping"

3



// - move() - prints "Animal is moving"

// - getAnimalInfo() - returns formatted animal details

}

// TODO: Create intermediate class Mammal extends Animal: public class Mammal extends Animal {

// TODO: Add mammal-specific fields:

// - furColor (String)

// - hasWarmBlood (boolean) - always true for mammals

// - gestationPeriod (int) - days

// TODO: Create constructor that:

// - Takes Animal parameters plus mammal-specific parameters // - Calls super() with appropriate parameters

// - Sets hasWarmBlood to true automatically

// - Prints "Mammal constructor: Adding mammal traits"

// TODO: Override methods from Animal:

// - Override move() to print "Mammal is walking/running" // - Call super.move() first, then add mammal-specific behavior

// TODO: Add mammal-specific methods:

// - nurse() - prints "Mammal is nursing offspring"

// - regulateTemperature() - prints "Maintaining body temperature" }

// TODO: Create specific class Dog extends Mammal:

public class Dog extends Mammal {

// TODO: Add dog-specific fields:

// - breed (String)

// - isDomesticated (boolean)

// - loyaltyLevel (int) - 1-10 scale

// - favoriteActivity (String)

// TODO: Create multiple constructors with different chaining patterns: // Constructor 1: Basic dog with minimal parameters

// - Calls super() with default mammal/animal values

// - Sets dog-specific defaults

// Constructor 2: Detailed dog with all parameters

4



// - Calls super() with all mammal/animal parameters

// - Initializes all dog-specific fields

// - Prints "Dog constructor: Creating [breed] dog"

// Constructor 3: Copy constructor

// - Takes another Dog object as parameter

// - Calls this() with parameters from source dog

// TODO: Override methods from the inheritance chain: // - Override eat() to show dog eating behavior

// - Call super.eat() and add "wagging tail while eating" // - Override move() to print "Dog is running and playing" // - Override sleep() to print "Dog is sleeping in doghouse"

// TODO: Add dog-specific methods:

// - bark() - prints "Woof! Woof!"

// - fetch() - prints "Dog is fetching the ball"

// - showLoyalty() - prints loyalty level message

// TODO: Create method that demonstrates calling up the chain: // - demonstrateInheritance() - calls methods from all three levels

public static void main(String[] args) {

// TODO: Test multilevel constructor chaining:

// 1. Create Dog object and observe all constructor calls in order // 2. Test different constructor patterns

// TODO: Test method overriding across levels:

// - Call same method and see how it's handled at each level // - Show how super calls propagate up the chain

// TODO: Test access to inherited members:

// - Access fields from all levels in hierarchy

// - Call methods from all levels

// TODO: Demonstrate the chain of inheritance:

// - Show Dog IS-A Mammal IS-A Animal relationships

// - Test instanceof operator with all levels

// TODO: Create multiple Dog objects with different constructor 5



patterns:

// - Show how constructor chaining works with different

initialization paths

}

}

PROGRAM:

// File: MultilevelInheritanceDemo.java

class Animal {

protected String species;

protected String habitat;

protected int lifespan;

protected boolean isWildlife;

public Animal(String species, String habitat, int lifespan, boolean isWildlife) {

this.species = species;

this.habitat = habitat;

this.lifespan = lifespan;

this.isWildlife = isWildlife;

System.out.println("Animal constructor: Creating " + species);

}

public void eat() {

System.out.println("Animal is eating");

}

public void sleep() {

System.out.println("Animal is sleeping");

}

public void move() {

System.out.println("Animal is moving");

}

public String getAnimalInfo() {

return String.format("Species: %s, Habitat: %s, Lifespan: %d years, Wildlife: %b",

species, habitat, lifespan, isWildlife);

}

}

class Mammal extends Animal {

protected String furColor;

protected boolean hasWarmBlood;

protected int gestationPeriod;

public Mammal(String species, String habitat, int lifespan, boolean isWildlife,

String furColor, int gestationPeriod) {

super(species, habitat, lifespan, isWildlife);

this.furColor = furColor;

this.gestationPeriod = gestationPeriod;

this.hasWarmBlood = true;

System.out.println("Mammal constructor: Adding mammal traits");

}

@Override

public void move() {

super.move();

System.out.println("Mammal is walking/running");

}

public void nurse() {

System.out.println("Mammal is nursing offspring");

}

public void regulateTemperature() {

System.out.println("Maintaining body temperature");

}

}

class Dog extends Mammal {

private String breed;

private boolean isDomesticated;

private int loyaltyLevel;

private String favoriteActivity;

public Dog() {

super("Dog", "Domestic", 13, false, "Brown", 60);

this.breed = "Mixed";

this.isDomesticated = true;

this.loyaltyLevel = 7;

this.favoriteActivity = "Playing";

System.out.println("Dog constructor: Creating " + breed + " dog");

}

public Dog(String species, String habitat, int lifespan, boolean isWildlife,

String furColor, int gestationPeriod, String breed,

boolean isDomesticated, int loyaltyLevel, String favoriteActivity) {

super(species, habitat, lifespan, isWildlife, furColor, gestationPeriod);

this.breed = breed;

this.isDomesticated = isDomesticated;

this.loyaltyLevel = loyaltyLevel;

this.favoriteActivity = favoriteActivity;

System.out.println("Dog constructor: Creating " + breed + " dog");

}

public Dog(Dog other) {

this(other.species, other.habitat, other.lifespan, other.isWildlife,

other.furColor, other.gestationPeriod, other.breed,

other.isDomesticated, other.loyaltyLevel, other.favoriteActivity);

}

@Override

public void eat() {

super.eat();

System.out.println("Dog is wagging tail while eating");

}

@Override

public void move() {

System.out.println("Dog is running and playing");

}

@Override

public void sleep() {

System.out.println("Dog is sleeping in doghouse");

}

public void bark() {

System.out.println("Woof! Woof!");

}

public void fetch() {

System.out.println("Dog is fetching the ball");

}

public void showLoyalty() {

System.out.println("Loyalty Level: " + loyaltyLevel + "/10");

}

public void demonstrateInheritance() {

System.out.println(getAnimalInfo());

move();

eat();

sleep();

nurse();

regulateTemperature();

bark();

fetch();

showLoyalty();

}

public static void main(String[] args) {

System.out.println("=== Test 1: Basic Dog Constructor ===");

Dog dog1 = new Dog();

dog1.demonstrateInheritance();

System.out.println("\n=== Test 2: Detailed Dog Constructor ===");

Dog dog2 = new Dog("Dog", "Domestic", 15, false, "Black", 65,

"Labrador", true, 9, "Swimming");

dog2.demonstrateInheritance();

System.out.println("\n=== Test 3: Copy Constructor ===");

Dog dog3 = new Dog(dog2);

dog3.demonstrateInheritance();

System.out.println("\n=== Test 4: instanceof checks ===");

System.out.println("dog1 instanceof Dog: " + (dog1 instanceof Dog));

System.out.println("dog1 instanceof Mammal: " + (dog1 instanceof Mammal));

System.out.println("dog1 instanceof Animal: " + (dog1 instanceof Animal));

}

}

OUTPUT:

=== Test 1: Basic Dog Constructor ===

Animal constructor: Creating Dog

Mammal constructor: Adding mammal traits

Dog constructor: Creating Mixed dog

Species: Dog, Habitat: Domestic, Lifespan: 13 years, Wildlife: false

Dog is running and playing

Animal is eating

Dog is wagging tail while eating

Dog is sleeping in doghouse

Mammal is nursing offspring

Maintaining body temperature

Woof! Woof!

Dog is fetching the ball

Loyalty Level: 7/10

=== Test 2: Detailed Dog Constructor ===

Animal constructor: Creating Dog

Mammal constructor: Adding mammal traits

Dog constructor: Creating Labrador dog

Species: Dog, Habitat: Domestic, Lifespan: 15 years, Wildlife: false

Dog is running and playing

Animal is eating

Dog is wagging tail while eating

Dog is sleeping in doghouse

Mammal is nursing offspring

Maintaining body temperature

Woof! Woof!

Dog is fetching the ball

Loyalty Level: 9/10

=== Test 3: Copy Constructor ===

Animal constructor: Creating Dog

Mammal constructor: Adding mammal traits

Dog constructor: Creating Labrador dog

Species: Dog, Habitat: Domestic, Lifespan: 15 years, Wildlife: false

Dog is running and playing

Animal is eating

Dog is wagging tail while eating

Dog is sleeping in doghouse

Mammal is nursing offspring

Maintaining body temperature

Woof! Woof!

Dog is fetching the ball

Loyalty Level: 9/10

=== Test 4: instanceof checks ===

dog1 instanceof Dog: true

dog1 instanceof Mammal: true

dog1 instanceof Animal: true

�� **PRACTICE PROBLEM 3: Hierarchical Inheritance with Method Overriding**

**Multiple classes extending the same parent with diverse overriding strategies** // File: HierarchicalInheritanceDemo.java

// TODO: Create base class Employee:

public class Employee {

// TODO: Create protected fields for inheritance:

// - employeeId (String)

// - name (String)

// - baseSalary (double)

// - department (String)

// - joiningDate (java.util.Date)

// TODO: Create constructor that:

// - Initializes all fields

// - Validates input parameters

// - Prints "Employee [name] created in [department]"

// TODO: Create methods that will be overridden differently by subclasses:

// - calculateSalary() - returns base salary (to be overridden) // - getJobDescription() - returns "General Employee" (to be overridden)

// - performWork() - prints "Employee is working" (to be overridden) // - attendMeeting() - prints "Employee attending meeting" (may be overridden)

// TODO: Create final methods that cannot be overridden:

6



// - getEmployeeId() - final method returning employee ID // - printEmployeeDetails() - final method showing all employee info

// TODO: Create methods that provide default behavior:

// - takeBreak() - standard break behavior for all employees // - clockIn() / clockOut() - standard time tracking

}

// TODO: Create Developer class extends Employee:

public class Developer extends Employee {

// TODO: Add developer-specific fields:

// - programmingLanguages (String[])

// - experienceLevel (String) - Junior/Mid/Senior

// - projectsCompleted (int)

// TODO: Create constructor using super:

// - Call super() with employee parameters

// - Initialize developer-specific fields

// - Print "Developer profile created"

// TODO: Override parent methods with @Override annotation: // @Override calculateSalary() - base salary + experience bonus + project bonus

// @Override getJobDescription() - return "Software Developer" // @Override performWork() - print "Developer is coding and debugging" // @Override attendMeeting() - print "Developer in technical meeting"

// TODO: Add developer-specific methods:

// - writeCode() - prints "Writing code in [language]"

// - reviewCode() - prints "Reviewing team's code"

// - deployApplication() - prints "Deploying application to production" }

// TODO: Create Manager class extends Employee:

public class Manager extends Employee {

// TODO: Add manager-specific fields:

// - teamSize (int)

// - managementLevel (String) - Team Lead/Manager/Director // - budgetResponsibility (double)

7



// TODO: Create constructor using super:

// - Call super() appropriately

// - Initialize manager fields

// - Print "Manager profile created"

// TODO: Override parent methods differently than Developer: // @Override calculateSalary() - base salary + team size bonus + management level bonus

// @Override getJobDescription() - return "Team Manager"

// @Override performWork() - print "Manager is coordinating team activities"

// @Override attendMeeting() - print "Manager leading strategic meeting"

// TODO: Add manager-specific methods:

// - conductPerformanceReview() - prints "Conducting team performance review"

// - assignTasks() - prints "Assigning tasks to team members" // - manageBudget() - prints "Managing department budget" }

// TODO: Create Intern class extends Employee:

public class Intern extends Employee {

// TODO: Add intern-specific fields:

// - university (String)

// - internshipDuration (int) - weeks

// - mentor (String) - mentor employee ID

// - isFullTime (boolean)

// TODO: Create constructor:

// - Call super() with modified parameters (lower salary, etc.) // - Initialize intern fields

// - Print "Intern onboarded"

// TODO: Override methods with intern-specific behavior:

// @Override calculateSalary() - return stipend amount (much lower than employees)

// @Override getJobDescription() - return "Intern"

// @Override performWork() - print "Intern is learning and assisting" // - DON'T override attendMeeting() - use parent implementation

8



// TODO: Add intern-specific methods:

// - attendTraining() - prints "Intern attending training session" // - submitReport() - prints "Submitting weekly progress report" // - seekMentorship() - prints "Getting guidance from mentor" }

public class HierarchicalInheritanceDemo {

public static void main(String[] args) {

// TODO: Create array of Employee references pointing to different subclass objects:

Employee[] employees = new Employee[4];

// TODO: Initialize with different employee types:

// - Create Developer, Manager, Intern objects

// - Store in Employee array (polymorphism)

// TODO: Demonstrate polymorphic method calls:

// - Loop through array calling same methods on different types // - Show how each subclass implements methods differently // - Call calculateSalary() and see different calculation logic

// TODO: Test @Override annotation benefits:

// - Show compile-time error detection

// - Demonstrate method signature matching

// TODO: Test method overriding vs method overloading:

// - Show overridden methods replace parent behavior

// - Show inherited methods work unchanged

// TODO: Demonstrate instanceof and type checking:

// - Check actual types of Employee references

// - Cast to specific types to access subclass-specific methods

// TODO: Test final method inheritance:

// - Show that final methods cannot be overridden

// - All subclasses inherit exact same behavior for final methods

// TODO: Create EmployeeManager utility class that:

// - Processes arrays of different employee types

9



// - Calculates total payroll using polymorphic calculateSalary() // - Generates reports showing different job descriptions // - Demonstrates benefits of hierarchical inheritance

}

}

**Key Learning Objectives for Each Problem: Problem 1 (Single Inheritance):**

● Understanding extends keyword and basic inheritance

● Constructor chaining with super() calls

● Method inheritance and overriding with @Override

● Access to protected fields in subclasses

● Polymorphic method calls

**Problem 2 (Multilevel Inheritance):**

● Building inheritance chains (Animal → Mammal → Dog)

● Constructor chaining across multiple levels

● Method overriding at different inheritance levels

● Using super.method() to call parent implementations

● Understanding IS-A relationships through the chain

**Problem 3 (Hierarchical Inheritance):**

● Multiple classes extending same parent (Employee → Developer, Manager, Intern) ● Different overriding strategies for same methods

● Polymorphic behavior with arrays of parent references

● @Override annotation benefits and compile-time checking

● Final methods that cannot be overridden

10