Core OOP Principles - Inheritance - Practice Problems - Any Two (1;3 done)

# 🛠 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

// - 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

// 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

}

}

import java.util.Random;

public class InheritanceDemo {

// Base class: Vehicle

static class Vehicle {

protected String brand;

protected String model;

protected int year;

protected String engineType;

private String registrationNumber;

private boolean isRunning;

// Default constructor

public Vehicle() {

this.brand = "Unknown Brand";

this.model = "Unknown Model";

this.year = 2000;

this.engineType = "Unknown Engine";

this.registrationNumber = generateRandomRegNumber();

this.isRunning = false;

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

}

// Parameterized constructor

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

this.brand = brand;

this.model = model;

this.year = year;

this.engineType = engineType;

this.registrationNumber = generateRandomRegNumber();

this.isRunning = false;

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

}

public void start() {

isRunning = true;

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

}

public void stop() {

isRunning = false;

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

}

public String getVehicleInfo() {

return "Brand: " + brand + ", Model: " + model + ", Year: " + year +

", Engine: " + engineType + ", Reg No: " + registrationNumber;

}

public void displaySpecs() {

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

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 regNo) {

this.registrationNumber = regNo;

}

public boolean isRunning() {

return isRunning;

}

private String generateRandomRegNumber() {

Random rand = new Random();

return "REG" + (1000 + rand.nextInt(9000));

}

}

// Derived class: Car

static class Car extends Vehicle {

private int numberOfDoors;

private String fuelType;

private String transmissionType;

// Default constructor

public Car() {

super();

this.numberOfDoors = 4;

this.fuelType = "Petrol";

this.transmissionType = "Manual";

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

}

// Parameterized constructor

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 sequence initiated");

}

@Override

public void displaySpecs() {

super.displaySpecs();

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

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

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

}

public void openTrunk() {

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

}

public void playRadio() {

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

}

}

// Main method to test everything

public static void main(String[] args) {

System.out.println("Creating Car using default constructor:");

Car car1 = new Car();

car1.displaySpecs();

car1.start();

car1.openTrunk();

car1.playRadio();

System.out.println();

System.out.println("Creating Car using parameterized constructor:");

Car car2 = new Car("Toyota", "Corolla", 2021, "V4", 4, "Diesel", "Automatic");

car2.displaySpecs();

car2.start();

car2.playRadio();

car2.openTrunk();

System.out.println();

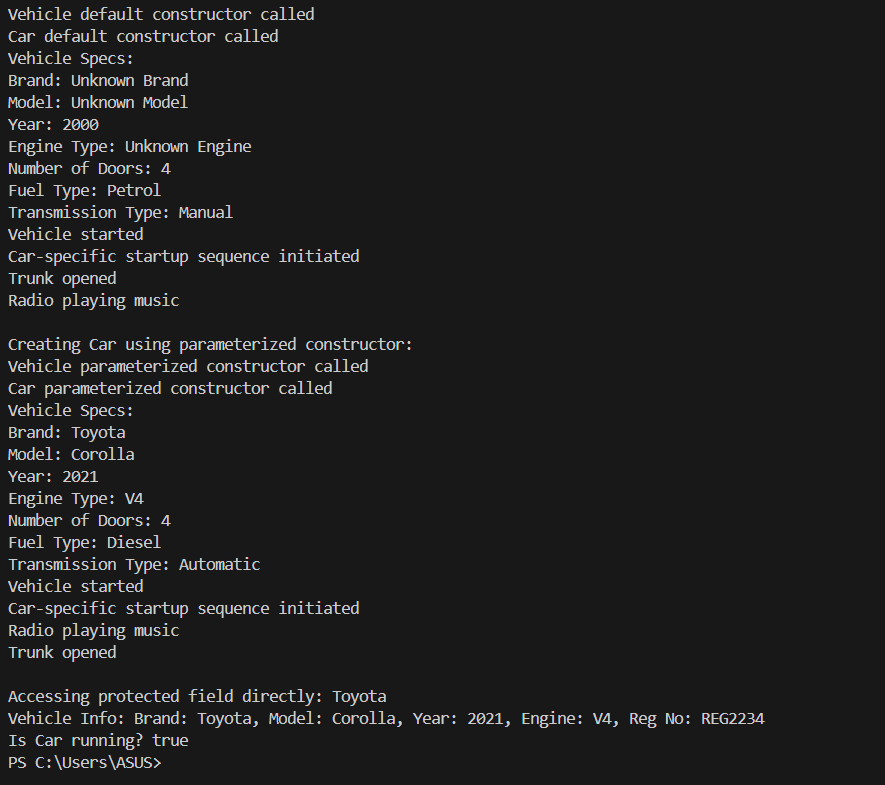
System.out.println("Accessing protected field directly: " + car2.brand);

System.out.println("Vehicle Info: " + car2.getVehicleInfo());

System.out.println("Is Car running? " + car2.isRunning());

}

}



# 🛠 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"

// - 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

// - 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



patterns:

// - Show how constructor chaining works with different initialization paths

}

}

# 🛠 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:

// - 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)



// 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

// 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



// - Calculates total payroll using polymorphic calculateSalary()

// - Generates reports showing different job descriptions

// - Demonstrates benefits of hierarchical inheritance

}

}

import java.util.Arrays;

class Employee {

private final String employeeId;

private String name;

private double baseSalary;

public Employee(String employeeId, String name, double baseSalary) {

this.employeeId = employeeId;

this.name = name;

this.baseSalary = baseSalary;

System.out.println("Employee profile created");

}

public final String getEmployeeId() {

return employeeId;

}

public final void printEmployeeDetails() {

System.out.println("ID: " + employeeId + ", Name: " + name + ", Base Salary: " + baseSalary);

}

public void takeBreak() {

System.out.println(name + " is taking a break");

}

public void clockIn() {

System.out.println(name + " clocked in");

}

public void clockOut() {

System.out.println(name + " clocked out");

}

public double calculateSalary() {

return baseSalary;

}

public String getJobDescription() {

return "General Employee";

}

public void performWork() {

System.out.println(name + " is working");

}

public void attendMeeting() {

System.out.println(name + " attending a meeting");

}

}

class Developer extends Employee {

private String[] programmingLanguages;

private String experienceLevel;

private int projectsCompleted;

public Developer(String employeeId, String name, double baseSalary,

String[] programmingLanguages, String experienceLevel, int projectsCompleted) {

super(employeeId, name, baseSalary);

this.programmingLanguages = programmingLanguages;

this.experienceLevel = experienceLevel;

this.projectsCompleted = projectsCompleted;

System.out.println("Developer profile created");

}

@Override

public double calculateSalary() {

return super.calculateSalary() + 1000 \* projectsCompleted;

}

@Override

public String getJobDescription() {

return "Software Developer";

}

@Override

public void performWork() {

System.out.println("Developer is coding and debugging");

}

@Override

public void attendMeeting() {

System.out.println("Developer in technical meeting");

}

public void writeCode() {

System.out.println("Writing code in " + String.join(", ", programmingLanguages));

}

public void reviewCode() {

System.out.println("Reviewing team's code");

}

public void deployApplication() {

System.out.println("Deploying application to production");

}

}

class Manager extends Employee {

private int teamSize;

private String managementLevel;

private double budgetResponsibility;

public Manager(String employeeId, String name, double baseSalary,

int teamSize, String managementLevel, double budgetResponsibility) {

super(employeeId, name, baseSalary);

this.teamSize = teamSize;

this.managementLevel = managementLevel;

this.budgetResponsibility = budgetResponsibility;

System.out.println("Manager profile created");

}

@Override

public double calculateSalary() {

return super.calculateSalary() + 500 \* teamSize;

}

@Override

public String getJobDescription() {

return "Team Manager";

}

@Override

public void performWork() {

System.out.println("Manager is coordinating team activities");

}

@Override

public void attendMeeting() {

System.out.println("Manager leading strategic meeting");

}

public void conductPerformanceReview() {

System.out.println("Conducting team performance review");

}

public void assignTasks() {

System.out.println("Assigning tasks to team members");

}

public void manageBudget() {

System.out.println("Managing department budget of $" + budgetResponsibility);

}

}

class Intern extends Employee {

private String university;

private int internshipDuration; // in weeks

private String mentorId;

private boolean isFullTime;

public Intern(String employeeId, String name, double stipend,

String university, int internshipDuration, String mentorId, boolean isFullTime) {

super(employeeId, name, stipend);

this.university = university;

this.internshipDuration = internshipDuration;

this.mentorId = mentorId;

this.isFullTime = isFullTime;

System.out.println("Intern onboarded");

}

@Override

public double calculateSalary() {

return super.calculateSalary(); // stipend

}

@Override

public String getJobDescription() {

return "Intern";

}

@Override

public void performWork() {

System.out.println("Intern is learning and assisting");

}

public void attendTraining() {

System.out.println("Intern attending training session");

}

public void submitReport() {

System.out.println("Submitting weekly progress report");

}

public void seekMentorship() {

System.out.println("Getting guidance from mentor " + mentorId);

}

}

public class HierarchicalInheritanceDemo {

public static void main(String[] args) {

Employee[] employees = new Employee[3];

employees[0] = new Developer("D001", "Alice", 5000,

new String[]{"Java", "Python"}, "Senior", 5);

employees[1] = new Manager("M001", "Bob", 7000,

10, "Director", 50000);

employees[2] = new Intern("I001", "Charlie", 1000,

"State University", 12, "M001", true);

for (Employee e : employees) {

e.printEmployeeDetails();

System.out.println("Job: " + e.getJobDescription());

e.performWork();

e.attendMeeting();

System.out.println("Calculated Salary: $" + e.calculateSalary());

e.takeBreak();

e.clockIn();

e.clockOut();

System.out.println();

if (e instanceof Developer) {

Developer dev = (Developer) e;

dev.writeCode();

dev.reviewCode();

dev.deployApplication();

} else if (e instanceof Manager) {

Manager mgr = (Manager) e;

mgr.conductPerformanceReview();

mgr.assignTasks();

mgr.manageBudget();

} else if (e instanceof Intern) {

Intern intern = (Intern) e;

intern.attendTraining();

intern.submitReport();

intern.seekMentorship();

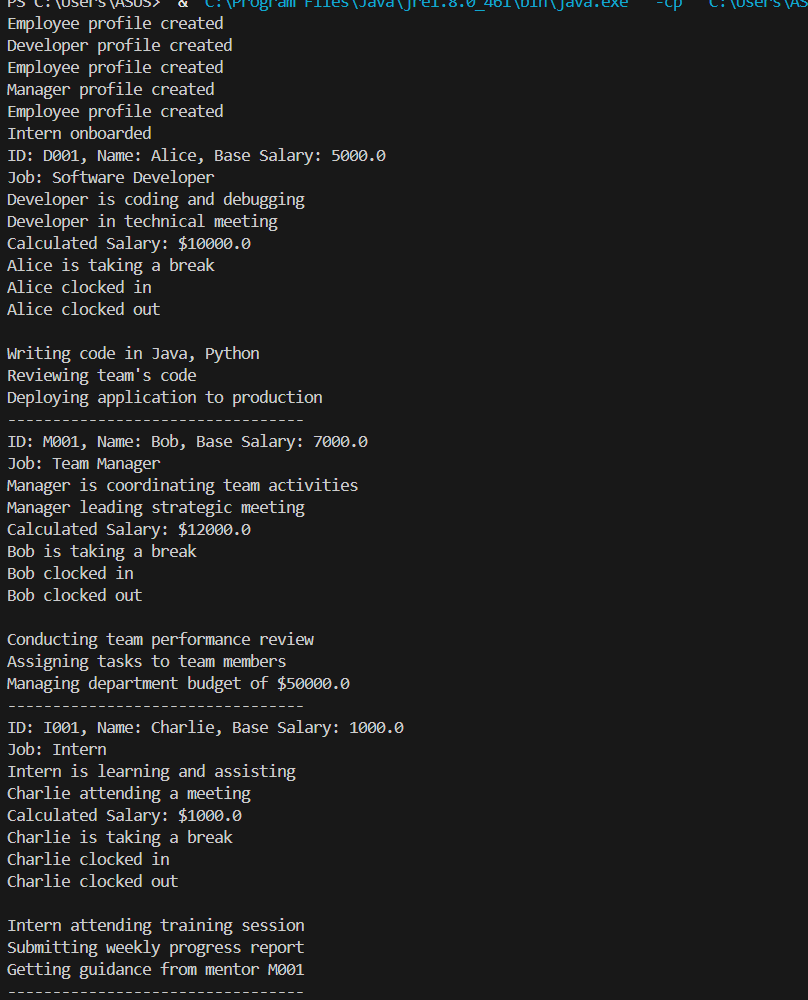
}

System.out.println("---------------------------------");

}

}

}



# 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