Enhanced Java OOP Practice Problems (Any 2) (1;2 done)

# Practice Problem 1: 🎮 Professional Virtual Pet Evolution System with Access Control

## Topics Covered: Access Modifiers, Encapsulation, JavaBean Standards, Constructor Chaining, final Keyword

**Requirements:** Design a VirtualPet class system that demonstrates professional Java development standards with proper access control, data hiding, and immutable configuration objects.

## Core Tasks:

1. Create VirtualPet class with four access levels:
   * private final String petId, PetSpecies species, long birthTimestamp (Immutable core)
   * private String petName, int age, happiness, health (Controlled mutable state)
   * protected static final String[] DEFAULT\_EVOLUTION\_STAGES (Package accessible)
   * static final int MAX\_HAPPINESS = 100, MAX\_HEALTH = 100

(Package-private constants)

* + public static final String PET\_SYSTEM\_VERSION = "2.0" (Global access)

1. Implement immutable PetSpecies class:
   * final class that cannot be extended
   * All fields must be final: speciesName, evolutionStages[], maxLifespan, habitat
   * Only getters (no setters) - defensive copying for arrays
   * Constructor validation with IllegalArgumentException for invalid data
2. Constructor chaining with validation:
   * Default constructor: Creates random pet with default species
   * Constructor with name only: Uses default species and moderate stats
   * Constructor with name and species: Uses default moderate stats
   * Main constructor: Full parameter validation, all others chain to this
3. JavaBean compliance:
   * Proper getter/setter naming conventions
   * Validated setters with range checking (0-100 for stats)
   * toString(), equals(), hashCode() implementations
   * Private helper methods: validateStat(), generatePetId(), checkEvolution()
4. Access-controlled methods:
   * public feedPet(String foodType), playWithPet(String gameType) - main interface
   * protected calculateFoodBonus(), calculateGameEffect() - internal calculations
   * private modifyHappiness(), modifyHealth(), updateEvolutionStage() - internal logic
   * Package-private getInternalState() for debugging within package
5. Create separate specialized pet classes (no inheritance):
   * DragonPet class with private final String dragonType, breathWeapon
   * RobotPet class with private boolean needsCharging, batteryLevel
   * Each class follows same access patterns and JavaBean standards
   * Use composition with VirtualPet reference if needed for shared functionality

import java.util.Arrays;

import java.util.Objects;

import java.util.UUID;

final class PetSpecies {

    private final String speciesName;

    private final String[] evolutionStages;

    private final int maxLifespan;

    private final String habitat;

    public PetSpecies(String speciesName, String[] evolutionStages, int maxLifespan, String habitat) {

        if (speciesName == null || speciesName.isEmpty()) throw new IllegalArgumentException("Invalid species name");

        if (evolutionStages == null || evolutionStages.length == 0) throw new

        IllegalArgumentException("Invalid evolution stages");

        if (maxLifespan <= 0) throw new IllegalArgumentException("Invalid lifespan");

        if (habitat == null || habitat.isEmpty()) throw new IllegalArgumentException("Invalid habitat");

        this.speciesName = speciesName;

        this.evolutionStages = Arrays.copyOf(evolutionStages, evolutionStages.length);

        this.maxLifespan = maxLifespan;

        this.habitat = habitat;

    }

    public String getSpeciesName() { return speciesName; }

    public String[] getEvolutionStages() { return Arrays.copyOf(evolutionStages, evolutionStages.length); }

    public int getMaxLifespan() { return maxLifespan; }

    public String getHabitat() { return habitat; }

    @Override

    public String toString() {

        return "Species: " + speciesName + ", Lifespan: " + maxLifespan + ", Habitat: " + habitat;

    }

}

class VirtualPet {

    private final String petId;

    private final PetSpecies species;

    private final long birthTimestamp;

    private String petName;

    private int age;

    private int happiness;

    private int health;

    private String currentStage;

    protected static final String[] DEFAULT\_EVOLUTION\_STAGES = {"Egg", "Baby", "Child", "Teen", "Adult", "Elder"};

    static final int MAX\_HAPPINESS = 100, MAX\_HEALTH = 100;

    public static final String PET\_SYSTEM\_VERSION = "2.0";

    public VirtualPet() {

        this("Mystery", new PetSpecies("Default", DEFAULT\_EVOLUTION\_STAGES, 80, "Forest"), 0, 50, 50);

    }

    public VirtualPet(String name) {

        this(name, new PetSpecies("Default", DEFAULT\_EVOLUTION\_STAGES, 80, "Forest"), 0, 60, 60);

    }

    public VirtualPet(String name, PetSpecies species) {

        this(name, species, 0, 70, 70);

    }

    public VirtualPet(String petName, PetSpecies species, int age, int happiness, int health) {

        this.petId = generatePetId();

        this.birthTimestamp = System.currentTimeMillis();

        this.species = species;

        setPetName(petName);

        setAge(age);

        setHappiness(happiness);

        setHealth(health);

        this.currentStage = species.getEvolutionStages()[0];

    }

    private String generatePetId() {

        return UUID.randomUUID().toString();

    }

    private int validateStat(int value) {

        if (value < 0) return 0;

        if (value > 100) return 100;

        return value;

    }

    private void updateEvolutionStage() {

        String[] stages = species.getEvolutionStages();

        int index = Math.min(age / 5, stages.length - 1);

        currentStage = stages[index];

    }

    public void feedPet(String foodType) {

        modifyHealth(calculateFoodBonus(foodType));

    }

    public void playWithPet(String gameType) {

        modifyHappiness(calculateGameEffect(gameType));

    }

    protected int calculateFoodBonus(String foodType) {

        return foodType.equalsIgnoreCase("fruit") ? 10 : 5;

    }

    protected int calculateGameEffect(String gameType) {

        return gameType.equalsIgnoreCase("ball") ? 15 : 8;

    }

    private void modifyHappiness(int value) {

        this.happiness = validateStat(this.happiness + value);

        updateEvolutionStage();

    }

    private void modifyHealth(int value) {

        this.health = validateStat(this.health + value);

        updateEvolutionStage();

    }

    String getInternalState() {

        return "Internal[ID=" + petId + ", Stage=" + currentStage + ", Age=" + age + "]";

    }

    public String getPetId() { return petId; }

    public PetSpecies getSpecies() { return species; }

    public long getBirthTimestamp() { return birthTimestamp; }

    public String getPetName() { return petName; }

    public int getAge() { return age; }

    public int getHappiness() { return happiness; }

    public int getHealth() { return health; }

    public String getCurrentStage() { return currentStage; }

    public void setPetName(String petName) {

        if (petName == null || petName.isEmpty()) this.petName = "Unnamed";

        else this.petName = petName;

    }

    public void setAge(int age) {

        this.age = Math.max(age, 0);

        updateEvolutionStage();

    }

    public void setHappiness(int happiness) {

        this.happiness = validateStat(happiness);

    }

    public void setHealth(int health) {

        this.health = validateStat(health);

    }

    @Override

    public String toString() {

        return "Pet[" + petName + ", Species=" + species.getSpeciesName() +

               ", Stage=" + currentStage + ", Age=" + age +

               ", Happiness=" + happiness + ", Health=" + health + "]";

    }

    @Override

    public boolean equals(Object o) {

        if (this == o) return true;

        if (!(o instanceof VirtualPet)) return false;

        VirtualPet pet = (VirtualPet) o;

        return petId.equals(pet.petId);

    }

    @Override

    public int hashCode() {

        return Objects.hash(petId);

    }

}

class DragonPet {

    private final String dragonType;

    private final String breathWeapon;

    private final VirtualPet basePet;

    public DragonPet(String name, String dragonType, String breathWeapon) {

        this.basePet = new VirtualPet(name);

        this.dragonType = dragonType;

        this.breathWeapon = breathWeapon;

    }

    public VirtualPet getBasePet() { return basePet; }

    public String getDragonType() { return dragonType; }

    public String getBreathWeapon() { return breathWeapon; }

    @Override

    public String toString() {

        return "DragonPet[" + basePet + ", Type=" + dragonType + ", Breath=" + breathWeapon + "]";

    }

}

class RobotPet {

    private boolean needsCharging;

    private int batteryLevel;

    private final VirtualPet basePet;

    public RobotPet(String name) {

        this.basePet = new VirtualPet(name);

        this.needsCharging = false;

        this.batteryLevel = 100;

    }

    public VirtualPet getBasePet() { return basePet; }

    public boolean isNeedsCharging() { return needsCharging; }

    public int getBatteryLevel() { return batteryLevel; }

    public void chargeBattery() {

        this.batteryLevel = 100;

        this.needsCharging = false;

    }

    @Override

    public String toString() {

        return "RobotPet[" + basePet + ", Battery=" + batteryLevel + ", Charging=" + needsCharging + "]";

    }

}

public class VirtualPetSystem {

    public static void main(String[] args) {

        PetSpecies dragonSpecies = new PetSpecies("Dragon",

                new String[]{"Egg", "Hatchling", "Wyrmling", "Adult", "Elder"},

                300, "Mountains");

        VirtualPet pet1 = new VirtualPet("Fluffy");

        VirtualPet pet2 = new VirtualPet("Smaug", dragonSpecies);

        pet1.feedPet("fruit");

        pet2.playWithPet("ball");

        DragonPet dp = new DragonPet("Draco", "Fire", "Flame Breath");

        RobotPet rp = new RobotPet("RoboBuddy");

        System.out.println(pet1);

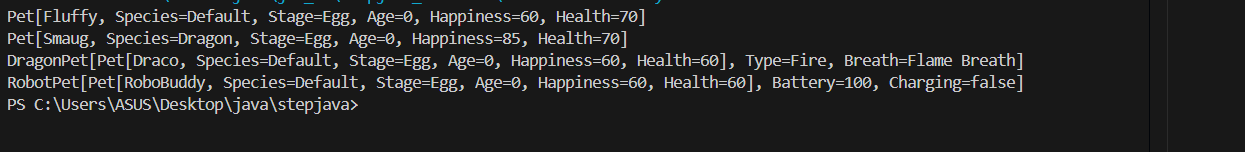
        System.out.println(pet2);

        System.out.println(dp);

        System.out.println(rp);

    }

}



# Practice Problem 2: 🏰 Medieval Kingdom Management with Security Architecture

## Topics Covered: Access Modifiers, Immutable Objects, instanceof, Constructor Overloading

**Requirements:** Build a magical kingdom system with proper access control and immutable configuration management using separate classes for different structure types.

## Core Tasks:

1. Create immutable KingdomConfig class:
   * final class with all final fields: kingdomName, foundingYear, allowedStructureTypes[], resourceLimits Map
   * Constructor with full validation and defensive copying
   * Only getters, no setters - return clones for mutable references
   * Factory methods: createDefaultKingdom(), createFromTemplate(String type)
2. Base MagicalStructure class (no inheritance hierarchy):
   * private final String structureId, long constructionTimestamp

(Immutable identity)

* + private final String structureName, location (Immutable properties)
  + private int magicPower, boolean isActive, String currentMaintainer (Controlled state)
  + static final int MIN\_MAGIC\_POWER = 0, MAX\_MAGIC\_POWER = 1000

(Package constants)

* + public static final String MAGIC\_SYSTEM\_VERSION = "3.0" (Global constant)

1. Constructor chaining in base class:
   * public MagicalStructure(String name, String location) - basic constructor
   * public MagicalStructure(String name, String location, int power) - with power
   * Main constructor: public MagicalStructure(String name, String location, int power, boolean active)
   * All constructors validate inputs and chain to main constructor
2. Four separate specialized structure classes:
   * WizardTower: private final int maxSpellCapacity, private List<String> knownSpells, private String currentWizard
   * EnchantedCastle: private final String castleType, private int defenseRating, private boolean hasDrawbridge
   * MysticLibrary: private final Map<String, String> bookCollection, private int knowledgeLevel
   * DragonLair: private final String dragonType, private long treasureValue, private int territorialRadius
3. Each structure class with unique constructor patterns:
   * WizardTower: Empty tower, basic spells, fully equipped options
   * Castle: Simple fort, royal castle, impregnable fortress variations
   * Library: Few books, moderate collection, ancient archives options
   * DragonLair: Different dragon types with specific lair requirements
4. KingdomManager class with instanceof usage:
   * private final List<Object> structures (stores different structure types)
   * private final KingdomConfig config
   * public static boolean canStructuresInteract(Object s1, Object s2)

- use instanceof for type checking

* + public static String performMagicBattle(Object attacker, Object defender)
  + public static int calculateKingdomPower(Object[] structures)
  + private String determineStructureCategory(Object structure) - type identification

1. JavaBean compliance for all classes:
   * Proper getter/setter patterns where appropriate
   * Immutable classes with only getters
   * Standard toString(), equals(), hashCode() methods

import java.util.\*;

final class KingdomConfig {

    private final String kingdomName;

    private final int foundingYear;

    private final String[] allowedTypes;

    private final Map<String, Integer> resourceLimits;

    private KingdomConfig(String name, int year, String[] types, Map<String, Integer> limits) {

        if (name == null || year <= 0) throw new IllegalArgumentException("Invalid config");

        this.kingdomName = name;

        this.foundingYear = year;

        this.allowedTypes = types.clone();

        this.resourceLimits = new HashMap<>(limits);

    }

    public static KingdomConfig createDefaultKingdom() {

        Map<String, Integer> resources = new HashMap<>();

        resources.put("Gold", 1000);

        resources.put("Mana", 500);

        return new KingdomConfig(

            "Avalon", 1066,

            new String[]{"Tower", "Castle", "Library", "Lair"},

            resources

        );

    }

    public String getKingdomName() { return kingdomName; }

    public int getFoundingYear() { return foundingYear; }

    public String[] getAllowedTypes() { return allowedTypes.clone(); }

    public Map<String, Integer> getResourceLimits() { return new HashMap<>(resourceLimits); }

}

class MagicalStructure {

    private final String id;

    private final long timestamp;

    private final String name;

    private final String location;

    private int magicPower;

    private boolean active;

    private String maintainer;

    public static final String MAGIC\_SYSTEM\_VERSION = "3.0";

    static final int MIN\_MAGIC\_POWER = 0, MAX\_MAGIC\_POWER = 1000;

    public MagicalStructure(String n, String l) {

        this(n, l, 100, true);

    }

    public MagicalStructure(String n, String l, int power) {

        this(n, l, power, true);

    }

    public MagicalStructure(String n, String l, int power, boolean a) {

        if (power < MIN\_MAGIC\_POWER || power > MAX\_MAGIC\_POWER) throw new IllegalArgumentException();

        this.id = UUID.randomUUID().toString();

        this.timestamp = System.currentTimeMillis();

        this.name = n;

        this.location = l;

        this.magicPower = power;

        this.active = a;

        this.maintainer = "None";

    }

    public String getName() { return name; }

    public int getMagicPower() { return magicPower; }

    public boolean isActive() { return active; }

    public String toString() { return name + "@" + location + " power=" + magicPower; }

}

// Specialized structures

class WizardTower {

    private final int maxSpellCapacity;

    private final List<String> knownSpells;

    private final String currentWizard;

    public WizardTower(int cap, List<String> spells, String wiz) {

        this.maxSpellCapacity = cap;

        this.knownSpells = new ArrayList<>(spells);

        this.currentWizard = wiz;

    }

    public String toString() { return "WizardTower of " + currentWizard; }

}

class EnchantedCastle {

    private final String castleType;

    private final int defenseRating;

    private final boolean hasDrawbridge;

    public EnchantedCastle(String t, int d, boolean draw) {

        this.castleType = t;

        this.defenseRating = d;

        this.hasDrawbridge = draw;

    }

    public String toString() { return "EnchantedCastle " + castleType; }

}

class MysticLibrary {

    private final Map<String, String> bookCollection;

    private final int knowledgeLevel;

    public MysticLibrary(Map<String, String> books, int level) {

        this.bookCollection = new HashMap<>(books);

        this.knowledgeLevel = level;

    }

    public String toString() { return "MysticLibrary level " + knowledgeLevel; }

}

class DragonLair {

    private final String dragonType;

    private final long treasureValue;

    private final int territorialRadius;

    public DragonLair(String type, long gold, int radius) {

        this.dragonType = type;

        this.treasureValue = gold;

        this.territorialRadius = radius;

    }

    public String toString() { return "DragonLair of " + dragonType; }

}

class KingdomManager {

    private final List<Object> structures = new ArrayList<>();

    private final KingdomConfig config;

    public KingdomManager(KingdomConfig cfg) { this.config = cfg; }

    public void addStructure(Object s) { structures.add(s); }

    public Object[] getStructures() { return structures.toArray(); }

    public static boolean canStructuresInteract(Object s1, Object s2) {

        return (s1 instanceof WizardTower && s2 instanceof MysticLibrary);

    }

    public static int calculateKingdomPower(Object[] arr) {

        return arr.length \* 100;

    }

}

// Demo

public class MedievalKingdomDemo {

    public static void main(String[] args) {

        KingdomConfig cfg = KingdomConfig.createDefaultKingdom();

        KingdomManager km = new KingdomManager(cfg);

        List<String> spells = new ArrayList<>();

        spells.add("Fireball");

        Map<String, String> books = new HashMap<>();

        books.put("Spellbook", "Fire");

        km.addStructure(new WizardTower(5, spells, "Merlin"));

        km.addStructure(new EnchantedCastle("Royal", 500, true));

        km.addStructure(new MysticLibrary(books, 300));

        km.addStructure(new DragonLair("Red", 10000, 50));

        System.out.println("Kingdom: " + cfg.getKingdomName());

        System.out.println("Power: " + KingdomManager.calculateKingdomPower(km.getStructures()));

    }

}

A black background with white text

AI-generated content may be incorrect.

# Practice Problem 3: 🚀 Space Station Security System with Final Attributes

## Topics Covered: final Keyword Variations, Access Modifiers, Immutable Security, Constructor Chaining

**Requirements:** Design a space station crew management system where security clearance levels are immutable and certain crew attributes cannot be changed.

## Core Tasks:

1. Immutable SecurityClearance class:
   * final class with private final String clearanceId, String level, String[] authorizedSections, long expirationDate
   * Constructor with validation, defensive copying for arrays
   * final methods that cannot be overridden: canAccess(String section), isExpired(), getAccessHash()
   * Only getters, return defensive copies for mutable references
2. CrewRank class with immutable structure:
   * final class with private final String rankName, int level, String[] permissions
   * Static factory methods: createCadet(), createOfficer(), createCommander(), createCaptain(), createAdmiral()
   * Only getters, no modification after creation
3. SpaceCrew class with mixed final/mutable attributes:
   * private final String crewId, homeplanet, SecurityClearance clearance (Immutable)
   * private final CrewRank initialRank (Starting position - never changes)
   * private CrewRank currentRank, int missionCount, double spaceHours

(Mutable with validation)

* + private boolean canAccessSection(String section) - uses final clearance
  + public static final String STATION\_NAME = "Stellar Odyssey"
  + public static final int MAX\_CREW\_CAPACITY = 50

1. Constructor chaining with security validation:
   * Emergency recruitment (minimal info, generates random homeplanet)
   * Standard recruitment (name, homeplanet, initialRank)
   * Experienced transfer (includes mission count and skills)
   * Full detailed constructor with security clearance
2. Specialized crew classes (separate, no inheritance):
   * CommandCrew class with private final Set<String> commandCertifications (Immutable certifications)
   * PilotCrew class with private final Set<String> flightCertifications

(Immutable certifications)

* + ScienceCrew class with private final String researchSpecialty

(Permanent specialization)

* + EngineerCrew class with private final String engineeringType (Final engineering type)

1. final methods for security:
   * public final String getCrewIdentification() - cannot be overridden
   * public final boolean canBePromoted() - security-critical logic
   * public final int calculateSecurityRating() - uses immutable attributes
   * private final boolean validateClearanceLevel() - internal security check
2. SpaceStationRegistry as final class:
   * final class that cannot be extended
   * private static final Map<String, Object> crewRegistry - station-wide tracking (stores different crew types)
   * public static boolean registerCrew(Object crew) - global crew management with instanceof checks
   * public static List<Object> getCrewByType(String type) - queries with access control
3. Access control scenarios:
   * Use instanceof to determine crew type for task assignment
   * Final security methods prevent tampering
   * Different access levels for emergency vs normal operations

# Practice Problem 4: 🎭 Advanced Story Generation with Immutable Character DNA

## Topics Covered: Complete Integration - All Concepts, Complex Object Interaction

**Requirements:** Create an AI-like story generator where characters have immutable genetic foundations but dynamic story development, demonstrating mastery of all OOP concepts.

## Core Tasks:

1. Immutable CharacterDNA system:
   * final class with private final String geneticId, personalityType, String[] innateTalents, characterArchetype
   * private final Map<String, Integer> baseAttributes - immutable foundation stats
   * Factory methods: createRandomDNA(), createFromTemplate(String template)
   * public final boolean isCompatibleWith(CharacterDNA other) - cannot be overridden
   * Only getters with defensive copying, no modification after creation
2. StoryCharacter class with complete access control:
   * private final String characterId, CharacterDNA dna, long birthTimestamp (Immutable core)
   * private String currentName, String currentLocation, String emotionalState (Dynamic state)
   * private final Map<String, String> relationships (Character bonds)
   * private int experiencePoints, Set<String> learnedSkills (Growth mechanics)
   * static final String STORY\_ENGINE\_VERSION = "4.0" (Package-private constant)
   * public static final String CHARACTER\_SYSTEM\_VERSION = "4.0" (Global version)
3. Constructor chaining with DNA validation:
   * public StoryCharacter(CharacterDNA dna) - minimal constructor
   * public StoryCharacter(CharacterDNA dna, String name) - with name
   * Main constructor: public StoryCharacter(CharacterDNA dna, String name, String startLocation, String mood)
   * Validate DNA compatibility with character requirements
4. Specialized character classes (separate classes):
   * HeroCharacter class: private final String destinyQuest, private String heroicVirtue, private Set<String> defeatedEnemies
   * VillainCharacter class: private final String evilScheme, private final String corruptionSource, private int evilInfluence
   * MysteriousCharacter class: Most attributes private, revealed through story progression
   * ComicCharacter class: private final String humorStyle, private int comedicTiming
5. Advanced constructor scenarios for each character type:
   * Characters from story prompts (parse text to determine DNA)
   * Random generation based on story genre requirements
   * Character creation with predefined personality templates
   * Import from previous stories with memory preservation
6. Complex instanceof story generation in StoryEngine class:
   * generateStoryArc(List<Object> characters) - different combinations create different plots using instanceof
   * resolveConflict(Object c1, Object c2) - interaction based on character types
   * createDialogue(Object character, String context) - speech patterns based on character type
   * determineStoryOutcome(List<Object> characters) - uses character DNA and relationships
7. Meta-story features with self-aware characters:
   * SelfAwareCharacter class that comments on own final limitations
   * Methods that attempt to modify immutable DNA (always fail, create humor)
   * Story commentary discussing access modifier restrictions
   * Characters that complain about encapsulation preventing direct attribute access
8. final class StoryEngine:
   * Cannot be extended, singleton pattern
   * private static final StoryEngine INSTANCE
   * private final Map<String, Object> activeCharacters (stores different character types)
   * private final String[] narrativeRules - immutable story laws
   * public final String generateNarrative() - cannot be overridden
   * Package-private methods for character registration and story mechanics
9. Complete JavaBean integration:
   * All classes follow strict getter/setter conventions
   * Immutable classes provide only getters with defensive copying
   * Builder pattern classes for complex character creation
   * Proper toString(), equals(), hashCode() for all story objects
10. Story serialization challenge:
    * Save/load story states while preserving final attribute integrity
    * Character compatibility matrix using instanceof for interaction rules
    * Achievement system tracking different constructor usage patterns
    * Story grammar system where character types determine available actions
11. Interactive elements:
    * User choices affect mutable character development but cannot change final DNA
    * Character memory system (stored in non-final fields)
    * Story branches based on character type combinations using instanceof
    * Real-time story statistics showing access modifier effects on character interaction