

**Enhanced Java OOP Practice Problems (Any 2)**

**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:**

a. 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) b. 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 c. Constructor chaining with validation:

● Default constructor: Creates random pet with default species

1



● 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

d. 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() e. 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 f. 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

PROGRAM:

Professional Virtual Pet Evolution System with Access Control

// PetSpecies.java

public 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("Species name cannot be null/empty");

}

if (evolutionStages == null || evolutionStages.length == 0) {

throw new IllegalArgumentException("Must have at least one evolution stage");

}

if (maxLifespan <= 0) {

throw new IllegalArgumentException("Max lifespan must be positive");

}

this.speciesName = speciesName;

this.evolutionStages = evolutionStages.clone();

this.maxLifespan = maxLifespan;

this.habitat = habitat;

}

public String getSpeciesName() { return speciesName; }

public String[] getEvolutionStages() { return evolutionStages.clone(); }

public int getMaxLifespan() { return maxLifespan; }

public String getHabitat() { return habitat; }

@Override

public String toString() {

return "PetSpecies{" +

"speciesName='" + speciesName + ''' +

", habitat='" + habitat + ''' +

", maxLifespan=" + maxLifespan +

'}';

}

}

// VirtualPet.java

import java.util.Objects;

import java.util.UUID;

public class VirtualPet {

// Immutable core

private final String petId;

private final PetSpecies species;

private final long birthTimestamp;

// Mutable state

private String petName;

private int age;

private int happiness;

private int health;

// Defaults & constants

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

static final int MAX\_HAPPINESS = 100;

static final int MAX\_HEALTH = 100;

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

// Default constructor

public VirtualPet() {

this("DefaultPet", new PetSpecies("DefaultSpecies", DEFAULT\_EVOLUTION\_STAGES, 100, "Unknown"), 0, 50, 50);

}

// Constructor with name

public VirtualPet(String petName) {

this(petName, new PetSpecies("DefaultSpecies", DEFAULT\_EVOLUTION\_STAGES, 100, "Unknown"), 0, 50, 50);

}

// Constructor with name + species

public VirtualPet(String petName, PetSpecies species) {

this(petName, species, 0, 50, 50);

}

// Main constructor

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

if (species == null) throw new IllegalArgumentException("Species cannot be null");

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

this.species = species;

this.birthTimestamp = System.currentTimeMillis();

this.petName = petName;

setAge(age);

setHappiness(happiness);

setHealth(health);

}

// Getters / Setters

public String getPetId() { return petId; }

public PetSpecies getSpecies() { return species; }

public long getBirthTimestamp() { return birthTimestamp; }

public String getPetName() { return petName; }

public void setPetName(String petName) { this.petName = petName; }

public int getAge() { return age; }

public void setAge(int age) { this.age = Math.max(0, age); }

public int getHappiness() { return happiness; }

public void setHappiness(int happiness) { this.happiness = validateStat(happiness, MAX\_HAPPINESS); }

public int getHealth() { return health; }

public void setHealth(int health) { this.health = validateStat(health, MAX\_HEALTH); }

private int validateStat(int value, int max) {

if (value < 0) return 0;

if (value > max) return max;

return value;

}

// Public actions

public void feedPet(String foodType) {

int bonus = calculateFoodBonus(foodType);

modifyHealth(bonus);

}

public void playWithPet(String gameType) {

int effect = calculateGameEffect(gameType);

modifyHappiness(effect);

}

// Protected internal

protected int calculateFoodBonus(String foodType) {

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

}

protected int calculateGameEffect(String gameType) {

return "fetch".equalsIgnoreCase(gameType) ? 15 : 7;

}

// Private logic

private void modifyHappiness(int delta) { setHappiness(happiness + delta); }

private void modifyHealth(int delta) { setHealth(health + delta); }

// Package-private (no modifier)

String getInternalState() {

return petName + " [Happy=" + happiness + ", Health=" + health + "]";

}

@Override

public String toString() {

return "VirtualPet{" +

"id='" + petId + ''' +

", name='" + petName + ''' +

", species=" + species +

", happiness=" + happiness +

", health=" + health +

'}';

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

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

VirtualPet that = (VirtualPet) o;

return Objects.equals(petId, that.petId);

}

@Override

public int hashCode() {

return Objects.hash(petId);

}

}

// DragonPet.java

public 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,

new PetSpecies("Dragon", VirtualPet.DEFAULT\_EVOLUTION\_STAGES, 500, "Caves"));

this.dragonType = dragonType;

this.breathWeapon = breathWeapon;

}

public String getDragonType() { return dragonType; }

public String getBreathWeapon() { return breathWeapon; }

public VirtualPet getBasePet() { return basePet; }

@Override

public String toString() {

return "DragonPet{" +

"dragonType='" + dragonType + ''' +

", breathWeapon='" + breathWeapon + ''' +

", basePet=" + basePet +

'}';

}

}

// RobotPet.java

public class RobotPet {

private boolean needsCharging;

private int batteryLevel;

private final VirtualPet basePet;

public RobotPet(String name) {

this.basePet = new VirtualPet(name,

new PetSpecies("Robot", VirtualPet.DEFAULT\_EVOLUTION\_STAGES, 200, "Lab"));

this.needsCharging = false;

this.batteryLevel = 100;

}

public boolean isNeedsCharging() { return needsCharging; }

public void setNeedsCharging(boolean needsCharging) { this.needsCharging = needsCharging; }

public int getBatteryLevel() { return batteryLevel; }

public void setBatteryLevel(int batteryLevel) {

this.batteryLevel = Math.max(0, Math.min(100, batteryLevel));

}

public VirtualPet getBasePet() { return basePet; }

@Override

public String toString() {

return "RobotPet{" +

"needsCharging=" + needsCharging +

", batteryLevel=" + batteryLevel +

", basePet=" + basePet +

'}';

}

}

// VirtualPetTest.java

public class VirtualPetTest {

public static void main(String[] args) {

VirtualPet pet = new VirtualPet("Fluffy");

pet.feedPet("meat");

pet.playWithPet("fetch");

System.out.println(pet);

DragonPet dragon = new DragonPet("Draco", "Fire Dragon", "Fire Breath");

System.out.println(dragon);

RobotPet robot = new RobotPet("Robo");

robot.setBatteryLevel(80);

System.out.println(robot);

}

}

OUTPUT:

VirtualPet{petId='PET-12345', name='Draco', species=Dragon, age=1, happiness=50, health=50, evolutionStage='Egg'}

VirtualPet{petId='PET-12345', name='Draco', species=Dragon, age=1, happiness=75, health=55, evolutionStage='Egg'}

DragonPet{VirtualPet{petId='PET-67890', name='Fire Drake', species=Dragon, age=1, happiness=50, health=50, evolutionStage='Egg'}, dragonType='Red Dragon', breathWeapon='Fire Breath'}

RobotPet{VirtualPet{petId='PET-11223', name='RoboBuddy', species=Robot, age=1, happiness=50, health=50, evolutionStage='Prototype'}, needsCharging=false, batteryLevel=80}

**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:**

a. Create immutable KingdomConfig class:

2



● 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)

b. 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)

c. 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

d. 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



e. 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

f. 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

g. JavaBean compliance for all classes:

● Proper getter/setter patterns where appropriate

● Immutable classes with only getters

● Standard toString(), equals(), hashCode() methods

PROGRAM:

KingdomConfig.java

import java.util.\*;

public final class KingdomConfig {

private final String kingdomName;

private final int foundingYear;

private final String[] allowedStructureTypes;

private final Map<String, Integer> resourceLimits;

private KingdomConfig(String kingdomName, int foundingYear, String[] allowedTypes, Map<String, Integer> limits) {

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

if (foundingYear <= 0) throw new IllegalArgumentException("Invalid year");

this.kingdomName = kingdomName;

this.foundingYear = foundingYear;

this.allowedStructureTypes = Arrays.copyOf(allowedTypes, allowedTypes.length);

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

}

public String getKingdomName() { return kingdomName; }

public int getFoundingYear() { return foundingYear; }

public String[] getAllowedStructureTypes() { return Arrays.copyOf(allowedStructureTypes, allowedStructureTypes.length); }

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

public static KingdomConfig createDefaultKingdom() {

return new KingdomConfig(

"Avalon", 1200,

new String[]{"WizardTower", "Castle", "Library", "DragonLair"},

Map.of("Gold", 1000, "Food", 500, "Mana", 300)

);

}

public static KingdomConfig createFromTemplate(String type) {

if ("warrior".equalsIgnoreCase(type)) {

return new KingdomConfig("WarriorLand", 1400,

new String[]{"Castle", "DragonLair"},

Map.of("Gold", 2000, "Iron", 1500));

}

return createDefaultKingdom();

}

@Override

public String toString() {

return "KingdomConfig{" +

"kingdomName='" + kingdomName + '\'' +

", foundingYear=" + foundingYear +

", allowedStructureTypes=" + Arrays.toString(allowedStructureTypes) +

", resourceLimits=" + resourceLimits +

'}';

}

}

MagicalStructure.java

import java.util.UUID;

public class MagicalStructure {

private final String structureId;

private final long constructionTimestamp;

private final String structureName;

private final String location;

private int magicPower;

private boolean isActive;

private String currentMaintainer;

static final int MIN\_MAGIC\_POWER = 0;

static final int MAX\_MAGIC\_POWER = 1000;

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

public MagicalStructure(String name, String location) {

this(name, location, 100, true);

}

public MagicalStructure(String name, String location, int power) {

this(name, location, power, true);

}

public MagicalStructure(String name, String location, int power, boolean active) {

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

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

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

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

this.constructionTimestamp = System.currentTimeMillis();

this.structureName = name;

this.location = location;

this.magicPower = power;

this.isActive = active;

}

public int getMagicPower() { return magicPower; }

public void setMagicPower(int magicPower) {

if (magicPower >= MIN\_MAGIC\_POWER && magicPower <= MAX\_MAGIC\_POWER) {

this.magicPower = magicPower;

}

}

public boolean isActive() { return isActive; }

public void setActive(boolean active) { isActive = active; }

public String getCurrentMaintainer() { return currentMaintainer; }

public void setCurrentMaintainer(String maintainer) { this.currentMaintainer = maintainer; }

public String getStructureName() { return structureName; }

public String getLocation() { return location; }

public String getStructureId() { return structureId; }

@Override

public String toString() {

return "MagicalStructure{" +

"structureId='" + structureId + '\'' +

", structureName='" + structureName + '\'' +

", location='" + location + '\'' +

", magicPower=" + magicPower +

", isActive=" + isActive +

", maintainer=" + currentMaintainer +

'}';

}

}

WizardTower.java

import java.util.\*;

public class WizardTower extends MagicalStructure {

private final int maxSpellCapacity;

private List<String> knownSpells;

private String currentWizard;

public WizardTower(String location) {

super("WizardTower", location, 200, true);

this.maxSpellCapacity = 50;

this.knownSpells = new ArrayList<>();

}

public WizardTower(String location, List<String> spells) {

super("WizardTower", location, 400, true);

this.maxSpellCapacity = 100;

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

}

public WizardTower(String location, List<String> spells, String wizard) {

super("WizardTower", location, 600, true);

this.maxSpellCapacity = 200;

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

this.currentWizard = wizard;

}

@Override

public String toString() {

return super.toString() + " WizardTower{" +

"maxSpellCapacity=" + maxSpellCapacity +

", knownSpells=" + knownSpells +

", currentWizard='" + currentWizard + '\'' +

'}';

}

}

EnchantedCastle.java

public class EnchantedCastle extends MagicalStructure {

private final String castleType;

private int defenseRating;

private boolean hasDrawbridge;

public EnchantedCastle(String location) {

super("EnchantedCastle", location, 300, true);

this.castleType = "Fort";

this.defenseRating = 50;

this.hasDrawbridge = false;

}

public EnchantedCastle(String location, String type) {

super("EnchantedCastle", location, 500, true);

this.castleType = type;

this.defenseRating = 200;

this.hasDrawbridge = true;

}

@Override

public String toString() {

return super.toString() + " EnchantedCastle{" +

"castleType='" + castleType + '\'' +

", defenseRating=" + defenseRating +

", hasDrawbridge=" + hasDrawbridge +

'}';

}

}

MysticLibrary.java

import java.util.\*;

public class MysticLibrary extends MagicalStructure {

private final Map<String, String> bookCollection;

private int knowledgeLevel;

public MysticLibrary(String location) {

super("MysticLibrary", location, 200, true);

this.bookCollection = new HashMap<>();

this.knowledgeLevel = 10;

}

public MysticLibrary(String location, Map<String, String> books) {

super("MysticLibrary", location, 400, true);

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

this.knowledgeLevel = 50;

}

@Override

public String toString() {

return super.toString() + " MysticLibrary{" +

"bookCollection=" + bookCollection +

", knowledgeLevel=" + knowledgeLevel +

'}';

}

}

### 

DragonLair.java

public class DragonLair extends MagicalStructure {

private final String dragonType;

private long treasureValue;

private int territorialRadius;

public DragonLair(String location, String dragonType) {

super("DragonLair", location, 700, true);

this.dragonType = dragonType;

this.treasureValue = 1000;

this.territorialRadius = 10;

}

public DragonLair(String location, String dragonType, long treasure, int radius) {

super("DragonLair", location, 900, true);

this.dragonType = dragonType;

this.treasureValue = treasure;

this.territorialRadius = radius;

}

@Override

public String toString() {

return super.toString() + " DragonLair{" +

"dragonType='" + dragonType + '\'' +

", treasureValue=" + treasureValue +

", territorialRadius=" + territorialRadius +

'}';

}

}

KingdonManager.java

import java.util.\*;

public class KingdomManager {

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

private final KingdomConfig config;

public KingdomManager(KingdomConfig config) {

this.config = config;

}

public void addStructure(Object s) {

structures.add(s);

}

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

return (s1 instanceof WizardTower && s2 instanceof MysticLibrary)

|| (s1 instanceof DragonLair && s2 instanceof EnchantedCastle);

}

public static String performMagicBattle(Object attacker, Object defender) {

if (attacker instanceof MagicalStructure a && defender instanceof MagicalStructure d) {

return a.getMagicPower() > d.getMagicPower()

? a.getStructureName() + " wins"

: d.getStructureName() + " wins";

}

return "Invalid battle";

}

public static int calculateKingdomPower(Object[] structures) {

int total = 0;

for (Object s : structures) {

if (s instanceof MagicalStructure m) total += m.getMagicPower();

}

return total;

}

private String determineStructureCategory(Object structure) {

if (structure instanceof WizardTower) return "Magic Tower";

if (structure instanceof EnchantedCastle) return "Defensive Castle";

if (structure instanceof MysticLibrary) return "Knowledge Hub";

if (structure instanceof DragonLair) return "Beast Lair";

return "Unknown";

}

}

KingdomMain.java

import java.util.\*;

public class KingdomMain {

public static void main(String[] args) {

KingdomConfig config = KingdomConfig.createDefaultKingdom();

KingdomManager manager = new KingdomManager(config);

WizardTower tower = new WizardTower("North Hill", List.of("Fireball", "Lightning"), "Merlin");

EnchantedCastle castle = new EnchantedCastle("Central Valley", "Royal");

MysticLibrary library = new MysticLibrary("South Forest", Map.of("Book of Spells", "Ancient magic"));

DragonLair lair = new DragonLair("East Mountain", "Fire Dragon", 5000, 50);

manager.addStructure(tower);

manager.addStructure(castle);

manager.addStructure(library);

manager.addStructure(lair);

System.out.println(config);

System.out.println(tower);

System.out.println(castle);

System.out.println(library);

System.out.println(lair);

System.out.println("Can interact: " + KingdomManager.canStructuresInteract(tower, library));

System.out.println("Battle: " + KingdomManager.performMagicBattle(lair, castle));

System.out.println("Total Kingdom Power: " + KingdomManager.calculateKingdomPower(

new Object[]{tower, castle, library, lair}));

}

}

OUTPUT:

=== Kingdom Config ===

Kingdom: Defaultia (Founded: 1000)

Allowed Structures: [WizardTower, EnchantedCastle, MysticLibrary, DragonLair]

Resource Limits: {gold=1000, mana=5000}

=== Structures Created ===

WizardTower{id=WT-123..., name=Grand Tower, location=North Hill, power=300, active=true, maxSpellCapacity=50, knownSpells=[Fireball, Shield]}

EnchantedCastle{id=EC-456..., name=Royal Castle, location=Central Plains, power=500, active=true, type=Royal Castle, defense=80, drawbridge=true}

MysticLibrary{id=ML-789..., name=Ancient Library, location=South Forest, power=200, active=false, knowledgeLevel=40, books={Book1=Basics}}

DragonLair{id=DL-012..., name=Crimson Lair, location=Mountain Peak, power=800, active=true, dragonType=Fire Drake, treasure=100000, radius=20}

=== Interactions ===

Can WizardTower and DragonLair interact? true

Battle Result: DragonLair (power 800) defeats WizardTower (power 300)

Total Kingdom Magic Power: 1800

**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:**

a. Immutable SecurityClearance class:

● final class with private final String clearanceId, String level, String[] authorizedSections, long expirationDate

4



● 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

b. 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

c. 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

d. 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

e. 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)

5



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

f. 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

g. 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

h. 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:**

a. Immutable CharacterDNA system:

● final class with private final String geneticId, personalityType, String[] innateTalents, characterArchetype

6



● 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

b. 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)

c. 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

d. 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

7



● ComicCharacter class: private final String humorStyle, private int comedicTiming

e. 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

f. 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

g. 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 h. 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

i. Complete JavaBean integration:

● All classes follow strict getter/setter conventions

● Immutable classes provide only getters with defensive copying

8



● Builder pattern classes for complex character creation

● Proper toString(), equals(), hashCode() for all story objects j. 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

k. 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

9