

**Java Constructor Practice Problems (Any Two) Best Programming Practice Guidelines**

1. Use Variables including for Fixed, User Inputs, and Results

2. Use Methods instead of writing code in the main() function

3. Proper naming conventions for all variables and methods

4. Proper Program Name and Class Name

5. Handle Checked and Unchecked Exceptions wherever possible

6. Proper Method Name which indicates action taking inputs and providing result

**Sample Program: Superhero Academy Registration System**

Create a program demonstrating constructor overloading and this() constructor chaining with superheroes:

// Program demonstrating superhero constructor chaining

import java.util.Scanner;

import java.util.Random;

class Superhero {

private String heroName;

private String realName;

private String superPower;

private int powerLevel;

private String origin;

private static int totalHeroes = 0;

private static final String ACADEMY\_NAME = "Xavier's Academy for Gifted Heroes";

// Default constructor - creates a random hero

public Superhero() {

this(generateRandomHeroName(), "Unknown Identity", "Undiscovered Power", 1, "Mysterious Origins");

}

// Constructor with hero name only

1



public Superhero(String heroName) {

this(heroName, "Secret Identity", "Discovering Powers", 10, "Training Origin");

}

// Constructor with hero name and power

public Superhero(String heroName, String superPower) {

this(heroName, "Classified", superPower, 25, "Acquired Powers"); }

// Full constructor

public Superhero(String heroName, String realName, String superPower, int powerLevel, String origin) {

this.heroName = heroName;

this.realName = realName;

this.superPower = superPower;

this.powerLevel = Math.max(1, Math.min(100, powerLevel)); // Clamp between 1-100

this.origin = origin;

totalHeroes++;

}

private static String generateRandomHeroName() {

String[] prefixes = {"Captain", "Super", "Wonder", "Mystic", "Shadow"};

String[] suffixes = {"Force", "Shield", "Blade", "Storm", "Fire"}; Random rand = new Random();

return prefixes[rand.nextInt(prefixes.length)] + " " +

suffixes[rand.nextInt(suffixes.length)];

}

public void displayHeroProfile() {

System.out.printf("�� Hero: %s | Real Name: %s | Power: %s | Level: %d | Origin: %s%n",

heroName, realName, superPower, powerLevel,

origin);

}

public static String getAcademyName() {

return ACADEMY\_NAME;

2



}

public static int getTotalHeroes() {

return totalHeroes;

}

}

**Practice Problem 1:** �� **Virtual Pet Evolution Simulator Topics Covered:** Constructor Overloading, this() Chaining, final Keyword, static Usage **Theme:** Create a Tamagotchi-style virtual pet that evolves based on care!

**Requirements:** Design a VirtualPet class that simulates pet evolution through different life stages.

**Hints:** a. Create VirtualPet class with fields:

● final String petId (generated using UUID-like system)

● String petName, String species, int age, int happiness, int health ● static final String[] EVOLUTION\_STAGES = {"Egg", "Baby", "Child", "Teen", "Adult", "Elder"}

● static int totalPetsCreated

b. Implement evolution-based constructors:

● Default constructor: Creates a mysterious egg with random species

● Constructor with name only: Pet starts as baby stage

● Constructor with name and species: Pet starts as child stage

● Full constructor: Specify all initial stats and stage

c. Use this() chaining where all constructors eventually call the main constructor d. Create unique methods:

● evolvePet(): Changes evolution stage based on age and care

● feedPet(), playWithPet(), healPet(): Affect happiness and health ● simulateDay(): Ages pet and randomly affects stats

● getPetStatus(): Returns current evolution stage

● static generatePetId(): Creates unique IDs

e. **Twist:** Pet dies if health reaches 0, becomes "Ghost" type that can't evolve but can haunt other pets!

f. In main method: Create a pet daycare with multiple pets, simulate several days, show evolution progress

SOLUTION:

import java.util.\*;

class VirtualPet {

private final String petId;

private String petName;

private String species;

private int age;

private int happiness;

private int health;

private String currentStage;

private boolean isGhost;

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

static int totalPetsCreated = 0;

public VirtualPet() {

this("Unknown", getRandomSpecies(), 0, 50, 50, EVOLUTION\_STAGES[0]);

}

public VirtualPet(String petName) {

this(petName, getRandomSpecies(), 0, 60, 60, EVOLUTION\_STAGES[1]);

}

public VirtualPet(String petName, String species) {

this(petName, species, 1, 70, 70, EVOLUTION\_STAGES[2]);

}

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

this.petId = generatePetId();

this.petName = petName;

this.species = species;

this.age = age;

this.happiness = happiness;

this.health = health;

this.currentStage = stage;

this.isGhost = false;

totalPetsCreated++;

}

public static String generatePetId() {

return UUID.randomUUID().toString();

}

private static String getRandomSpecies() {

String[] speciesList = {"Dragon", "Cat", "Dog", "Phoenix", "Slime"};

return speciesList[new Random().nextInt(speciesList.length)];

}

public void feedPet() {

if (!isGhost) {

health = Math.min(100, health + 10);

happiness += 5;

}

}

public void playWithPet() {

if (!isGhost) {

happiness = Math.min(100, happiness + 10);

health -= 5;

}

}

public void healPet() {

if (!isGhost) {

health = Math.min(100, health + 15);

}

}

public void evolvePet() {

if (isGhost) return;

int stageIndex = Arrays.asList(EVOLUTION\_STAGES).indexOf(currentStage);

if (age > 2 && stageIndex < EVOLUTION\_STAGES.length - 1) {

currentStage = EVOLUTION\_STAGES[stageIndex + 1];

}

}

public void simulateDay() {

if (isGhost) return;

age++;

happiness -= new Random().nextInt(6);

health -= new Random().nextInt(6);

if (health <= 0) {

currentStage = "Ghost";

isGhost = true;

} else {

evolvePet();

}

}

public String getPetStatus() {

return String.format("[%s] %s (%s) - Age: %d, Health: %d, Happiness: %d, Stage: %s",

petId.substring(0, 8), petName, species, age, health, happiness, currentStage);

}

}

public class PetDayCare {

public static void main(String[] args) {

List<VirtualPet> pets = new ArrayList<>();

pets.add(new VirtualPet());

pets.add(new VirtualPet("Fluffy"));

pets.add(new VirtualPet("Rex", "Dragon"));

pets.add(new VirtualPet("Mimi", "Cat", 5, 80, 90, "Teen"));

for (int day = 1; day <= 5; day++) {

System.out.println("\n--- Day " + day + " ---");

for (VirtualPet pet : pets) {

pet.simulateDay();

System.out.println(pet.getPetStatus());

}

}

System.out.println("\nTotal pets created: " + VirtualPet.totalPetsCreated);

}

}

OUTPUT:

--- Day 1 ---

[2d4a3b04] Unknown (Dragon) - Age: 1, Health: 48, Happiness: 50, Stage: Egg

[ff286b84] Fluffy (Cat) - Age: 1, Health: 60, Happiness: 59, Stage: Baby

[cc39ce75] Rex (Dragon) - Age: 2, Health: 66, Happiness: 67, Stage: Child

[e79a5e61] Mimi (Cat) - Age: 6, Health: 85, Happiness: 76, Stage: Adult

--- Day 2 ---

[2d4a3b04] Unknown (Dragon) - Age: 2, Health: 45, Happiness: 46, Stage: Egg

[ff286b84] Fluffy (Cat) - Age: 2, Health: 58, Happiness: 57, Stage: Baby

[cc39ce75] Rex (Dragon) - Age: 3, Health: 61, Happiness: 62, Stage: Teen

[e79a5e61] Mimi (Cat) - Age: 7, Health: 84, Happiness: 72, Stage: Elder

--- Day 3 ---

[2d4a3b04] Unknown (Dragon) - Age: 3, Health: 42, Happiness: 43, Stage: Baby

[ff286b84] Fluffy (Cat) - Age: 3, Health: 57, Happiness: 52, Stage: Child

[cc39ce75] Rex (Dragon) - Age: 4, Health: 58, Happiness: 60, Stage: Adult

[e79a5e61] Mimi (Cat) - Age: 8, Health: 81, Happiness: 71, Stage: Elder

--- Day 4 ---

[2d4a3b04] Unknown (Dragon) - Age: 4, Health: 40, Happiness: 40, Stage: Child

[ff286b84] Fluffy (Cat) - Age: 4, Health: 53, Happiness: 51, Stage: Teen

[cc39ce75] Rex (Dragon) - Age: 5, Health: 56, Happiness: 55, Stage: Elder

[e79a5e61] Mimi (Cat) - Age: 9, Health: 79, Happiness: 71, Stage: Elder

--- Day 5 ---

[2d4a3b04] Unknown (Dragon) - Age: 5, Health: 36, Happiness: 39, Stage: Teen

er

[e79a5e61] Mimi (Cat) - Age: 10, Health: 78, Happiness: 67, Stage: Elder

Total pets created: 4

3



**Practice Problem 2:** �� **Medieval Kingdom Builder with Magic System Topics Covered:** instanceof Type Checking, Inheritance, Constructor Chaining, this Keyword **Theme:** Build a magical kingdom where different structures have unique powers!

**Requirements:** Create a kingdom building system with magical structures that interact with each other.

**Hints:** a. Create abstract MagicalStructure base class:

● Fields: String structureName, int magicPower, String location, boolean isActive

● Constructor overloading with this() chaining

● Abstract method: castMagicSpell()

b. Create derived magical structures:

● WizardTower (additional: int spellCapacity, String[] knownSpells) ● EnchantedCastle (additional: int defenseRating, boolean hasDrawbridge) ● MysticLibrary (additional: int bookCount, String ancientLanguage) ● DragonLair (additional: String dragonType, int treasureValue)

c. Each structure type has unique constructor patterns:

● WizardTower: Can be built empty, with basic spells, or fully equipped ● Castle: Can be a simple fort, royal castle, or impregnable fortress

● Library: Can start with few books or ancient collections

● DragonLair: Different dragons have different lair requirements

d. Implement magical interactions using instanceof:

4



● static boolean canStructuresInteract(MagicalStructure s1, MagicalStructure s2)

● static String performMagicBattle(MagicalStructure attacker, MagicalStructure defender)

● static int calculateKingdomMagicPower(MagicalStructure[] structures)

e. **Twist:** Some structure combinations create special effects:

● WizardTower + Library = Knowledge boost (double spell capacity)

● Castle + DragonLair = Dragon guard (triple defense)

● Multiple towers = Magic network (shared spell pool)

f. Create a KingdomManager that uses instanceof to:

● Categorize structures by type

● Calculate different tax rates for each structure type

● Determine kingdom specialization (Magic-focused, Defense-focused, etc.)

SOLUTION:

import java.util.\*;

abstract class MagicalStructure {

protected String structureName;

protected int magicPower;

protected String location;

protected boolean isActive;

public MagicalStructure() {

this("Unknown Structure", 0, "Unknown", true);

}

public MagicalStructure(String structureName) {

this(structureName, 0, "Unknown", true);

}

public MagicalStructure(String structureName, int magicPower, String location, boolean isActive) {

this.structureName = structureName;

this.magicPower = magicPower;

this.location = location;

this.isActive = isActive;

}

public abstract String castMagicSpell();

}

class WizardTower extends MagicalStructure {

private int spellCapacity;

private String[] knownSpells;

public WizardTower() {

this("Wizard Tower", 100, "Hilltop", true, 5, new String[]{"Fireball"});

}

public WizardTower(String[] spells) {

this("Wizard Tower", 150, "Mountain", true, spells.length, spells);

}

public WizardTower(String structureName, int magicPower, String location, boolean isActive, int spellCapacity, String[] knownSpells) {

super(structureName, magicPower, location, isActive);

this.spellCapacity = spellCapacity;

this.knownSpells = knownSpells;

}

public String castMagicSpell() {

return structureName + " casts " + (knownSpells.length > 0 ? knownSpells[0] : "a mysterious spell");

}

public int getSpellCapacity() {

return spellCapacity;

}

public void doubleSpellCapacity() {

this.spellCapacity \*= 2;

}

}

class EnchantedCastle extends MagicalStructure {

private int defenseRating;

private boolean hasDrawbridge;

public EnchantedCastle() {

this("Simple Fort", 50, "Valley", true, 100, false);

}

public EnchantedCastle(String name, boolean royal) {

this(name, royal ? 200 : 100, "Plains", true, royal ? 300 : 150, true);

}

public EnchantedCastle(String structureName, int magicPower, String location, boolean isActive, int defenseRating, boolean hasDrawbridge) {

super(structureName, magicPower, location, isActive);

this.defenseRating = defenseRating;

this.hasDrawbridge = hasDrawbridge;

}

public String castMagicSpell() {

return structureName + " radiates protective wards.";

}

public void tripleDefense() {

this.defenseRating \*= 3;

}

public int getDefenseRating() {

return defenseRating;

}

}

class MysticLibrary extends MagicalStructure {

private int bookCount;

private String ancientLanguage;

public MysticLibrary() {

this("Small Library", 30, "Village", true, 100, "Latin");

}

public MysticLibrary(int books) {

this("Grand Library", 120, "Capital", true, books, "Sanskrit");

}

public MysticLibrary(String structureName, int magicPower, String location, boolean isActive, int bookCount, String ancientLanguage) {

super(structureName, magicPower, location, isActive);

this.bookCount = bookCount;

this.ancientLanguage = ancientLanguage;

}

public String castMagicSpell() {

return structureName + " whispers knowledge in " + ancientLanguage;

}

}

class DragonLair extends MagicalStructure {

private String dragonType;

private int treasureValue;

public DragonLair() {

this("Dragon Lair", 200, "Cave", true, "Fire Dragon", 5000);

}

public DragonLair(String dragonType) {

this("Dragon Lair", 300, "Mountain Cave", true, dragonType, 8000);

}

public DragonLair(String structureName, int magicPower, String location, boolean isActive, String dragonType, int treasureValue) {

super(structureName, magicPower, location, isActive);

this.dragonType = dragonType;

this.treasureValue = treasureValue;

}

public String castMagicSpell() {

return dragonType + " roars and unleashes fiery breath.";

}

}

class StructureInteractions {

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

return !(s1 instanceof DragonLair && s2 instanceof MysticLibrary);

}

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

if (attacker.magicPower > defender.magicPower) {

return attacker.structureName + " defeats " + defender.structureName;

} else if (attacker.magicPower < defender.magicPower) {

return defender.structureName + " withstands " + attacker.structureName;

} else {

return "Both structures are evenly matched";

}

}

public static int calculateKingdomMagicPower(MagicalStructure[] structures) {

int total = 0;

for (MagicalStructure s : structures) {

total += s.magicPower;

}

return total;

}

public static void applySpecialEffects(MagicalStructure[] structures) {

for (MagicalStructure s1 : structures) {

for (MagicalStructure s2 : structures) {

if (s1 instanceof WizardTower && s2 instanceof MysticLibrary) {

((WizardTower) s1).doubleSpellCapacity();

}

if (s1 instanceof EnchantedCastle && s2 instanceof DragonLair) {

((EnchantedCastle) s1).tripleDefense();

}

}

}

}

}

class KingdomManager {

public static void categorizeStructures(MagicalStructure[] structures) {

for (MagicalStructure s : structures) {

if (s instanceof WizardTower) System.out.println(s.structureName + " is a WizardTower");

else if (s instanceof EnchantedCastle) System.out.println(s.structureName + " is a Castle");

else if (s instanceof MysticLibrary) System.out.println(s.structureName + " is a Library");

else if (s instanceof DragonLair) System.out.println(s.structureName + " is a DragonLair");

}

}

public static void calculateTax(MagicalStructure[] structures) {

for (MagicalStructure s : structures) {

int tax;

if (s instanceof WizardTower) tax = 100;

else if (s instanceof EnchantedCastle) tax = 200;

else if (s instanceof MysticLibrary) tax = 50;

else if (s instanceof DragonLair) tax = 500;

else tax = 0;

System.out.println(s.structureName + " pays tax: " + tax);

}

}

public static void determineSpecialization(MagicalStructure[] structures) {

int magic = 0, defense = 0;

for (MagicalStructure s : structures) {

if (s instanceof WizardTower || s instanceof MysticLibrary) magic++;

if (s instanceof EnchantedCastle || s instanceof DragonLair) defense++;

}

if (magic > defense) System.out.println("Kingdom is Magic-focused");

else if (defense > magic) System.out.println("Kingdom is Defense-focused");

else System.out.println("Kingdom is Balanced");

}

}

public class MedievalKingdom {

public static void main(String[] args) {

MagicalStructure[] structures = {

new WizardTower(),

new EnchantedCastle(),

new MysticLibrary(),

new DragonLair()

};

KingdomManager.categorizeStructures(structures);

KingdomManager.calculateTax(structures);

StructureInteractions.applySpecialEffects(structures);

System.out.println("Total Magic Power: " + StructureInteractions.calculateKingdomMagicPower(structures));

KingdomManager.determineSpecialization(structures);

System.out.println(StructureInteractions.performMagicBattle(structures[0], structures[1]));

}

}

OUTPUT:

Wizard Tower is a WizardTower

Simple Fort is a Castle

Small Library is a Library

Dragon Lair is a DragonLair

Wizard Tower pays tax: 100

Simple Fort pays tax: 200

Small Library pays tax: 50

Dragon Lair pays tax: 500

Total Magic Power: 380

Kingdom is Balanced

Wizard Tower defeats Simple Fort

**Practice Problem 3:** �� **Space Station Crew Management System Topics Covered:** final Keyword Variations, static Usage, Constructor Overloading, this Keyword **Theme:** Manage a space station where crew members have fixed roles but evolving skills!

**Requirements:** Design a space crew system where certain attributes are permanently fixed while others can change.

**Hints:** a. Create SpaceCrew class with strategic final usage:

● final String crewId (cannot change - permanent space ID)

● final String homeplanet (where they're from - cannot change) ● final CrewRank initialRank (starting rank - promotional history) ● Regular fields: currentRank, skillLevel, missionCount, spaceHours ● static final String STATION\_NAME = "Stellar Odyssey" ● static final int MAX\_CREW\_CAPACITY = 50

b. Create final enum CrewRank:

5

public enum CrewRank {



CADET(1), OFFICER(2), COMMANDER(3), CAPTAIN(4), ADMIRAL(5); private final int level;

// Constructor and methods

}

c. Implement diverse constructors:

● Emergency recruitment (minimal info - generates random homeplanet) ● Standard recruitment (name, homeplanet, rank)

● Experienced transfer (includes previous mission count and skills)

● Full detailed profile constructor

d. Create final methods that cannot be overridden:

● final String getCrewIdentification(): Returns permanent ID info ● final boolean canBePromoted(): Based on fixed initial rank and current status ● final int calculateSpaceExperience(): Complex calculation that subclasses cannot modify

e. **Unique twist:** Create specialized crew types:

● PilotCrew (cannot change flight certifications once assigned)

● ScienceCrew (research specialization is permanent)

● EngineerCrew (engineering type certification is final)

f. Create a final class SpaceStationRegistry (cannot be extended):

● Manages all crew assignments

● Has static methods for station-wide operations

● Tracks crew statistics and handles emergencies

g. **Space Emergency Scenario:** When station faces crisis, certain crew combinations are needed:

● Use final methods to determine eligibility

● Some crew members' final attributes make them irreplaceable for certain tasks 6



**Practice Problem 4:** �� **Interactive Story Generator with Dynamic Characters**

**Topics Covered:** All Concepts Integration - The Ultimate Challenge!

**Theme:** Create an AI-like story generator where characters evolve and interact dynamically!

**Requirements:** Build a complex story system where characters have fixed backstories but dynamic relationships and adventures.

**Hints:** a. Create StoryCharacter base class with mixed final and dynamic attributes:

● final String characterId, final String backstory, final PersonalityType corePersonality

● Dynamic: currentMood, relationshipMap, skillSet, currentLocation ● static final String[] STORY\_GENRES = {"Fantasy", "Sci-Fi", "Mystery", "Romance", "Adventure"}

b. Character types with unique constructor patterns:

● Hero class: Different origin stories determine final abilities

● Villain class: Evil motivation is permanent, but methods can evolve ● MysteriousStranger class: Most attributes hidden, revealed through story ● Comic Relief class: Humor style is final, but timing is dynamic

c. Advanced constructor chaining scenarios:

● Characters can be created from story prompts (parse text to determine traits) ● Characters can be generated randomly based on genre

● Characters can be imported from previous stories (with memory of past adventures) ● Characters can be created through "character fusion" (combining two existing characters)

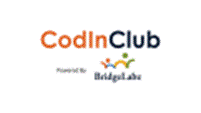
d. Complex instanceof usage for story generation:

● generateStoryArc(): Different character type combinations create different plot types

● resolveConflict(): How characters interact depends on their exact types ● createDialogue(): Speech patterns determined by character inheritance hierarchy

e. **Ultimate twist - Meta-Story Features:**

7



● Characters can become "self-aware" and comment on their final limitations ● Story can break "fourth wall" where characters complain about their constructor limitations

● Characters can attempt to "hack" their own final attributes (always fails, but creates humor)

f. Multi-layered story mechanics:

● StoryEngine class uses all constructor concepts

● Characters have final destinies but dynamic paths to reach them

● Stories can be saved/loaded, preserving final attributes while allowing character growth

● Story generation uses static methods for universal story rules

g. **Interactive Elements:**

● User choices affect character development (but can't change final traits) ● Characters remember past interactions (stored in non-final fields)

● Story branches based on character type combinations

● Achievement system tracking different constructor usage patterns

h. **Technical Challenges:**

● Implement story serialization (saving character states while preserving final integrity) ● Create a character relationship matrix using instanceof for compatibility ● Build a story grammar system where character types determine available story actions ● Design character evolution paths that respect final limitations while maximizing growth

**Expected Unique Output:**

● Generated stories that change based on character constructor combinations ● Character dialogue that reflects their constructor heritage

● Interactive story choices that demonstrate the constructor concept understanding ● Meta-commentary where characters discuss their own programming limitations ● Story statistics showing how different constructor patterns affect narrative outcomes

8