Java Constructor Practice Problems (Any Two)

3;4 done

## 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



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;

}

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

1. 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
2. Use this() chaining where all constructors eventually call the main constructor
3. 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
4. **Twist:** Pet dies if health reaches 0, becomes "Ghost" type that can't evolve but can haunt other pets!
5. In main method: Create a pet daycare with multiple pets, simulate several days, show evolution progress

# 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()

1. 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)
2. 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
3. Implement magical interactions using instanceof:
   * static boolean canStructuresInteract(MagicalStructure s1, MagicalStructure s2)
   * static String performMagicBattle(MagicalStructure attacker, MagicalStructure defender)
   * static int calculateKingdomMagicPower(MagicalStructure[] structures)
4. **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)
5. 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.)

# 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

1. Create final enum CrewRank:

public enum CrewRank {

CADET(1), OFFICER(2), COMMANDER(3), CAPTAIN(4), ADMIRAL(5);

private final int level;

// Constructor and methods

}

1. 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
2. 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
3. **Unique twist:** Create specialized crew types:
   * PilotCrew (cannot change flight certifications once assigned)
   * ScienceCrew (research specialization is permanent)
   * EngineerCrew (engineering type certification is final)
4. 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
5. **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

Sol:

enum CrewRank { CADET, OFFICER, COMMANDER, CAPTAIN, ADMIRAL }

public class SpaceCrew {

final String crewId;

final String homePlanet;

final CrewRank initialRank;

CrewRank currentRank;

int skillLevel, missionCount, spaceHours;

static final String STATION\_NAME = "Stellar Odyssey";

static final int MAX\_CREW\_CAPACITY = 50;

SpaceCrew(String id, String planet, CrewRank rank) {

crewId = id;

homePlanet = planet;

initialRank = rank;

currentRank = rank;

}

final String getCrewIdentification() {

return crewId + " from " + homePlanet;

}

final boolean canBePromoted() {

return initialRank.ordinal() < CrewRank.ADMIRAL.ordinal();

}

final int calculateSpaceExperience() {

return missionCount \* spaceHours + skillLevel;

}

public static void main(String[] args) {

SpaceCrew member = new SpaceCrew("ID123", "Mars", CrewRank.OFFICER);

member.missionCount = 5;

member.spaceHours = 1000;

member.skillLevel = 50;

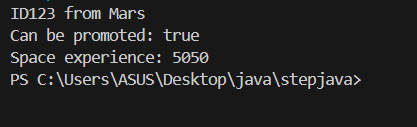
System.out.println(member.getCrewIdentification());

System.out.println("Can be promoted: " + member.canBePromoted());

System.out.println("Space experience: " + member.calculateSpaceExperience());

}

}



# 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"}

1. 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
2. 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)
3. 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

## Ultimate twist - Meta-Story Features:

* + 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)

1. 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

## 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

## 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

enum PersonalityType { HEROIC, EVIL, MYSTERIOUS, FUNNY }

public class StoryCharacter {

final String characterId;

final String backstory;

final PersonalityType corePersonality;

String currentMood;

String currentLocation;

StoryCharacter(String id, String backstory, PersonalityType personality) {

characterId = id;

this.backstory = backstory;

corePersonality = personality;

}

void generateStoryArc() {

System.out.println("Character " + characterId + " embarks on an adventure.");

}

void resolveConflict() {

System.out.println("Character " + characterId + " resolves a conflict.");

}

void createDialogue() {

System.out.println("Character says: 'This is my story.'");

}

public static void main(String[] args) {

StoryCharacter hero = new StoryCharacter("H001", "Born in village, destined for greatness.", PersonalityType.HEROIC);

hero.generateStoryArc();

hero.resolveConflict();

hero.createDialogue();

}

}

