POLYMORPHISM PRACTICE PROBLEMS (1;3 done)

**PRACTICE PROBLEM 1: Gaming Arena - Method Overloading**

**Understanding compile-time polymorphism through method overloading in a gaming context.**

// File: GameBattle.java public class GameBattle {

// TODO: Create an 'attack' method that takes damage (int) and prints "Basic attack for [damage] points!"

public void attack(int damage) {

// ... (implementation)

}

// TODO: Overload 'attack' method to take damage and weapon name

// Print "Attacking with [weapon] for [damage] points!" public void attack(int damage, String weapon) {

// ... (implementation)

}

// TODO: Overload 'attack' method for critical hits (damage, weapon, isCritical)

// If critical: "CRITICAL HIT! [weapon] deals [damage\*2] points!"

// Else: use the previous overloaded method

public void attack(int damage, String weapon, boolean isCritical)

{

// ... (implementation)

}

// TODO: Overload 'attack' for team attacks (damage, String[] teammates)

// Print "Team attack with [teammate names] for [damage \* team size] total damage!"

public void attack(int damage, String[] teammates) {

// ... (implementation)

}

public static void main(String[] args) {

// TODO: Gaming Battle Simulation:

// 1. Create a GameBattle object

// 2. Test all overloaded attack methods:

// - Basic attack with 50 damage

// - Sword attack with 75 damage

// - Critical bow attack with 60 damage

// - Team attack with {"Alice", "Bob"} for 40 base damage

// 3. Observe how the compiler chooses the correct method based on parameters

}

}

**Sol: // File: GameBattle.java**

**public class GameBattle {**

**public void attack(int damage) {**

**System.out.println("Basic attack for " + damage + " points!");**

**}**

**public void attack(int damage, String weapon) {**

**System.out.println("Attacking with " + weapon + " for " + damage + " points!");**

**}**

**public void attack(int damage, String weapon, boolean isCritical) {**

**if (isCritical) {**

**System.out.println("CRITICAL HIT! " + weapon + " deals " + (damage \* 2) + " points!");**

**} else {**

**attack(damage, weapon);**

**}**

**}**

**public void attack(int damage, String[] teammates) {**

**int totalDamage = damage \* teammates.length;**

**System.out.print("Team attack with ");**

**for (String mate : teammates) System.out.print(mate + " ");**

**System.out.println("for " + totalDamage + " total damage!");**

**}**

**public static void main(String[] args) {**

**GameBattle g = new GameBattle();**

**g.attack(50);**

**g.attack(75, "Sword");**

**g.attack(60, "Bow", true);**

**g.attack(40, new String[]{"Alice", "Bob"});**

**}**

**}**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

# PRACTICE PROBLEM 2: Social Media Platform - Method Overriding

### Demonstrating runtime polymorphism through method overriding in social media context.

// File: SocialMediaDemo.java public class SocialMediaPost {

protected String content; protected String author;

public SocialMediaPost(String content, String author) { this.content = content;

this.author = author;

}

// TODO: Create 'share()' method that prints "Sharing: [content] by [author]"



public void share() {

// ... (implementation)

}

}

public class InstagramPost extends SocialMediaPost { private int likes;

public InstagramPost(String content, String author, int likes) { super(content, author);

this.likes = likes;

}

// TODO: Override 'share()' to print " Instagram: [content] by @[author] - [likes] likes "

@Override

public void share() {

// ... (implementation)

}

}

public class TwitterPost extends SocialMediaPost { private int retweets;

public TwitterPost(String content, String author, int retweets) { super(content, author);

this.retweets = retweets;

}

// TODO: Override 'share()' to print "Tweet: [content] by @[author] - [retweets] retweets "

@Override

public void share() {

// ... (implementation)

}

}



public class SocialMediaDemo {

public static void main(String[] args) {

// TODO: Social Media Feed Simulation:

// 1. Create array of SocialMediaPost references

// 2. Add InstagramPost("Sunset vibes!", "john\_doe", 245)

// 3. Add TwitterPost("Java is awesome!", "code\_ninja", 89)

// 4. Add regular SocialMediaPost("Hello world!", "beginner")

// 5. Loop through and call share() on each - observe different behaviors!

}

}

# PRACTICE PROBLEM 3: Food Delivery App - Dynamic Method Dispatch

### Exploring how JVM resolves method calls at runtime based on actual object type.

// File: FoodDelivery.java public class Restaurant {

protected String name;

public Restaurant(String name) { this.name = name;

}

// TODO: Create 'prepareFood()' method that prints "[name] is preparing generic food"

public void prepareFood() {

// ... (implementation)

}

// TODO: Create 'estimateTime()' method that prints "Estimated



time: 30 minutes"

public void estimateTime() {

// ... (implementation)

}

}

public class PizzaPlace extends Restaurant { public PizzaPlace(String name) {

super(name);

}

// TODO: Override 'prepareFood()' to print " [name] is making delicious pizza with fresh toppings!"

@Override

public void prepareFood() {

// ... (implementation)

}

// TODO: Override 'estimateTime()' to print "Pizza ready in 20 minutes! "

@Override

public void estimateTime() {

// ... (implementation)

}

}

public class SushiBar extends Restaurant { public SushiBar(String name) {

super(name);

}

// TODO: Override 'prepareFood()' to print "[name] is crafting fresh sushi with precision!"

@Override

public void prepareFood() {

// ... (implementation)

}



// TODO: Override 'estimateTime()' to print "Sushi will be ready in 25 minutes! "

@Override

public void estimateTime() {

// ... (implementation)

}

}

public class FoodDelivery {

public static void main(String[] args) {

// TODO: Dynamic Food Ordering System:

// 1. Create a Restaurant reference variable

// 2. Assign new PizzaPlace("Mario's Pizza") to it

// 3. Call prepareFood() and estimateTime() - observe Pizza methods execute

// 4. Reassign to new SushiBar("Tokyo Sushi")

// 5. Call same methods again - observe Sushi methods execute

// 6. Explain how JVM knows which method to call at runtime!

}

}

// File: FoodDelivery.java

class Restaurant {

protected String name;

public Restaurant(String name) {

this.name = name;

}

public void prepareFood() {

System.out.println(name + " is preparing generic food");

}

public void estimateTime() {

System.out.println("Estimated time: 30 minutes");

}

}

class PizzaPlace extends Restaurant {

public PizzaPlace(String name) { super(name); }

@Override

public void prepareFood() {

System.out.println(name + " is making delicious pizza with fresh toppings!");

}

@Override

public void estimateTime() {

System.out.println("Pizza ready in 20 minutes!");

}

}

class SushiBar extends Restaurant {

public SushiBar(String name) { super(name); }

@Override

public void prepareFood() {

System.out.println(name + " is crafting fresh sushi with precision!");

}

@Override

public void estimateTime() {

System.out.println("Sushi will be ready in 25 minutes!");

}

}

public class FoodDelivery {

public static void main(String[] args) {

Restaurant r = new PizzaPlace("Mario's Pizza");

r.prepareFood();

r.estimateTime();

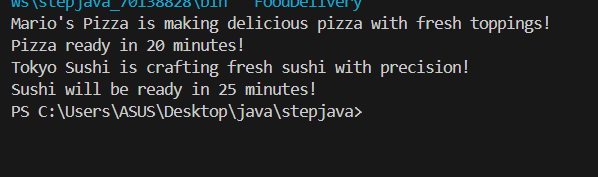
r = new SushiBar("Tokyo Sushi");

r.prepareFood();

r.estimateTime();

}

}



# PRACTICE PROBLEM 4: University System - Upcasting Adventures

### Learning safe upcasting and accessing inherited members in university context.

// File: UniversitySystem.java public class Person {

protected String name; protected int age;

protected String email;

public Person(String name, int age, String email) { this.name = name;

this.age = age;

this.email = email;

}

// TODO: Create 'introduce()' method that prints "Hi! I'm [name], [age] years old."

public void introduce() {

// ... (implementation)

}

// TODO: Create 'getContactInfo()' that prints "Email: [email]" public void getContactInfo() {

// ... (implementation)

}

}

public class Student extends Person { private String studentId;

private String major;

public Student(String name, int age, String email, String studentId, String major) {

super(name, age, email);

this.studentId = studentId; this.major = major;

}

// TODO: Create 'attendLecture()' method that prints "[name] is attending [major] lecture"

public void attendLecture() {

// ... (implementation)

}

// TODO: Create 'submitAssignment()' that prints "Assignment submitted by [studentId]"

public void submitAssignment() {

// ... (implementation)

}

}

public class Professor extends Person { private String department;

public Professor(String name, int age, String email, String department) {

super(name, age, email);

this.department = department;

}

// TODO: Create 'conductClass()' that prints "Prof. [name] is teaching [department]"

public void conductClass() {

// ... (implementation)

}

}

public class UniversitySystem {

public static void main(String[] args) {

// TODO: University Registration Demo:

// 1. Create Student("Alice", 20, ["alice@uni.edu",](mailto:alice@uni.edu) "CS2021", "Computer Science")

// 2. Upcast Student to Person reference: Person p = new Student(...)

// 3. Call introduce() and getContactInfo() using Person reference

// 4. Try calling attendLecture() with Person reference - observe compile error

// 5. Access the 'name' field through Person reference

// 6. Explain why upcasting is safe but limits access to subclass-specific methods



}

}

# PRACTICE PROBLEM 5: Entertainment System - Mastering Downcasting

### Learning explicit downcasting to access subclass-specific functionality.

// File: EntertainmentHub.java public class Entertainment {

protected String title;

public Entertainment(String title) { this.title = title;

}

// TODO: Create 'start()' method that prints "Starting [title]" public void start() {

// ... (implementation)

}

// TODO: Create 'stop()' method that prints "Stopping [title]" public void stop() {

// ... (implementation)

}

}

public class Movie extends Entertainment { private String genre;

public Movie(String title, String genre) {



super(title);

this.genre = genre;

}

// TODO: Create 'showSubtitles()' that prints "Showing subtitles for [title] ([genre])"

public void showSubtitles() {

// ... (implementation)

}

// TODO: Create 'adjustQuality()' that prints "Adjusting video quality for [title]"

public void adjustQuality() {

// ... (implementation)

}

}

public class Game extends Entertainment { private String platform;

public Game(String title, String platform) { super(title);

this.platform = platform;

}

// TODO: Create 'saveProgress()' that prints "Saving [title] progress on [platform]"

public void saveProgress() {

// ... (implementation)

}

// TODO: Create 'showLeaderboard()' that prints "[title] leaderboard on [platform]"

public void showLeaderboard() {

// ... (implementation)

}

}

public class EntertainmentHub {

public static void main(String[] args) {

// TODO: Entertainment Center Simulation:

// 1. Create Entertainment reference and assign new Movie("Avengers", "Action")

// 2. Call start() through Entertainment reference

// 3. Downcast Entertainment reference to Movie: Movie m = (Movie) entertainment;

// 4. Call showSubtitles() and adjustQuality() using Movie reference

// 5. Repeat with Game("FIFA 24", "PlayStation")

// 6. Demonstrate what happens if you try wrong downcast (Movie to Game)

}

}

# PRACTICE PROBLEM 6: Smart Home - Safe Downcasting with instanceof

### Mastering the instanceof operator for safe type checking and downcasting.

// File: SmartHome.java public class SmartDevice {

# Learning Objectives Summary

## Problem 1 - Method Overloading (Compile-time Polymorphism)

* Understanding method signatures and parameter differences
* How compiler resolves overloaded methods at compile time
* Practical application in gaming scenarios

## Problem 2 - Method Overriding (Runtime Polymorphism)

* Overriding inherited methods in subclasses
* Using @Override annotation for clarity
* Social media context makes concepts relatable

## Problem 3 - Dynamic Method Dispatch

* How JVM resolves method calls at runtime
* Understanding the difference between reference type and object type
* Real-world food delivery scenario

## Problem 4 - Upcasting

* Safe conversion from subclass to superclass reference
* Accessing inherited members through superclass reference
* Understanding compile-time method binding limitations

## Problem 5 - Downcasting

* Explicit casting from superclass to subclass reference
* Accessing subclass-specific methods and fields
* Understanding potential ClassCastException risks

## Problem 6 - Safe Downcasting with instanceof

* Using instanceof operator for type checking
* Preventing runtime exceptions through safe downcasting
* Writing robust and flexible polymorphic code

# Challenge Tips

* **Start Simple**: Begin with basic implementations, then add creative touches
* **Test Everything**: Try different scenarios to see polymorphism in action
* **Think Real-World**: These problems simulate actual software scenarios
* **Debug Creatively**: If something doesn't work, experiment with different approaches
* **Time Goal**: Aim to complete all 6 problems within 60 minutes!

**Happy Coding!**