Object Oriented Programming with Java - Teamwork Project

Prepared by: Dr. Azhar Alhindi

Weight: 10%

Deadline: Week 12 (April 26th - 30th, 2026), before your scheduled lab session.

Instructions:

• This is a teamwork project; every three students must form a group.

• Each group must submit one project by the leader.

• All submitted projects must be unique. Any duplicated work will receive a grade of 0.

• Projects must be submitted by the deadline; late submissions will not be accepted and will receive a grade of 0.

• Every student must attend an oral discussion on their project. Failure to attend will result in a grade of 0 for the discussion part.

• The discussion grade may vary for each student based on their individual performance.

• Submit a single compressed (zipped) folder containing all your `.java` files via the E-learning platform. The folder should be named as [ID1_ID2_ID3_Names].

Project Title: Amazon-Style E-Commerce Management System

Project Overview

In this project, you will develop a Java-based backend system for an organization (like Amazon) that manages **Products** and **Orders**. The goal is to apply advanced Java concepts to create a scalable, organized, and robust application.

Phase 1: The Foundations (Data & Interfaces)

Before building the "logic," you need the "blueprints."

1. **Create the financial Interface (Calculatable):** Define two methods:
    - double calculateDiscount(): Determines the reduction in price based on specific product rules.
    - double calculateTax(double rate): Logic to calculate tax based on the current price and a provided percentage.
2. **Create the Product Abstract Class:**
    - Implement Calculatable and Comparable<Product>.
    - **Fields**: int id, String name, double price.

- **Logic: @Override** compareTo() to return this.price - other.price.
- **Logic: @Override** equals() to return true only if the id matches.
- **Methods:**

- **@Override** calculateTax () to *return* only the tax amount.
  For example: Returns the tax amount: **price * (rate / 100)**.
- Add an abstract copy() method, deep copy. This forces every subclass to define how it should be duplicated.
- Add toString() method to provide a clear, human-readable summary of the product's current attributes.

Phase 2: Inheritance & Customization

Now, specialize your products into two distinct categories.

- **Subclasses <u>Electronics</u> & <u>Clothing</u>**:-
  - **Unique Attributes:** Electronics adds warrantyMonths; Clothing adds size.
  - **Constructor Logic:** Both must utilize **super(id, name, price)** to initialize inherited fields.
  - **Polymorphic Implementation:**

- **@Override** calculateDiscount(): Apply a 10% reduction for Electronics and 20% for Clothing and finally, return the reduction amount.
- **@Override** copy(): Implement deep cloning to return a new instance of the specific subclass.
- **@Override** toString(): Expand to include both parent and subclass-specific data.

Phase 3: The Order System & Polymorphism

This is where inventory management happens.

- **Define the Order Class:**
  - **Array of Objects**: Use a Product[] inventory to store up to 10 items.
  - **Static vs. Instance**:
    1. Each Order has an orderID (**instance**).
    2. The system must track totalOrdersCreated (**static**) which increases every time a new Order object is instantiated.
  - **Polymorphism**: Implement a method that iterates through the Product[] array. It should utilize **polymorphism** by calling the toString(), calculateDiscount(), and calculateTax() methods on each item, regardless of whether it is an Electronics or Clothing object.
  - **addProduct(Product p)**: a method to add one product at a time to the array, use the copy() method to ensure the order has its own unique instance.

Ensure the addProduct method checks if there is space before adding a new item. Also, use the equals() method you defined in Phase 1 to ensure that no two products with the same ID are added to the same Order instance.

Phase 4: Professional Standards

- **Encapsulation**: No class variables should be public. Use private with appropriate getters/setters.
- **Documentation- Javadoc:** You must document every class and method using the @param, @return, and @author tags.

Phase 5: Testing and Validation

The Main.java entry point must demonstrate system capabilities: -

1- Create items/instances of Electronics and Clothing.
2- Create a Product array, populate it with items of different prices, and sort them by price. Finally, display the sorted list.
3- Create two orders and add a **Laptop** to the first. Add the same product to the second order using a **Deep Copy** to ensure unique memory instances for each. Finally, add a different second product to the second order to differentiate them further.

   Note: If you change the price of the laptop in Order 1, the laptop in Order 2 should remain unchanged. This proves your copy() method works!
   - You can use *order1.getProduct(0).equals(order2.getProduct(0))* to prove they are logically the same item (same ID) and you can then use the == operator to show they are *different* memory addresses.
   - The **Benefit:** to understand the difference between **Logical Equality** (via equals()) and **Reference Equality** (via ==).

4- Loop through your products and use instanceof to calculate specific insurance fees and display the results:
   - if (item instanceof Electronics) : add 5% fee.
   - if (item instanceof Clothing) : add 0% fee.
5- Generate the final order summaries by invoking the printOrderDetails() method for each instance/order.

Project Deliverables

1. **The Source Code**: All .java files.
2. **Documentation**: A generated Javadoc folder.

Sample Output:

```
--- Catalog Sorted by Price (Low to High) ---
ID: 201 | Vintage Tee | Price: $40.00 | Size: L
ID: 202 | Denim Jeans | Price: $60.00 | Size: 32W
ID: 102 | Smartphone | Price: $800.00 | Warranty: 12m
ID: 101 | Gaming Laptop | Price: $1500.00 | Warranty: 24m

--- Applied Insurance Fees ---
[Clothing] Vintage Tee Fee: $0.00
[Clothing] Denim Jeans Fee: $0.00
[Electronics] Smartphone Fee: $40.0
[Electronics] Gaming Laptop Fee: $75.0

--- Order Summary (ID: 1) ---
ID: 101 | Gaming Laptop | Price: $1500.00 | Warranty: 24m
   Discount: -$150.00 | Tax: +$75.00

--- Order Summary (ID: 2) ---
ID: 101 | Gaming Laptop | Price: $1500.00 | Warranty: 24m
   Discount: -$150.00 | Tax: +$75.00
ID: 202 | Denim Jeans | Price: $60.00 | Size: 32W
   Discount: -$12.00 | Tax: +$3.00
```

Project Rubric (10% Total)

Discussion/Oral Test (2%)

- Each student will be evaluated individually during the discussion.
- Grades may vary between students based on their individual performance.
- If a student does not attend the discussion, they will receive 0 for this part.

Criteria	Beginning (0 pts)	Emerging (50% pts)	Proficient (100% pts)
Code Structure & Readability (1%)	Poorly organized; difficult to read; lacks comments.	Somewhat organized; inconsistent indentation or comments.	Code is well-organized, properly indented, and includes clear comments.
Classes & Relationships (2%)	Missing classes or incorrect relationships (e.g., Order/Product).	Most classes implemented with minor logic mistakes.	All classes (Order, Product, etc.) implemented with correct relationships.
Encapsulation & Data Hiding (1%)	Variables are public; no getters or setters used.	Some variables private; accessors/mutators are incomplete.	All variables private; full use of proper setters and getters.
Deep Copy Logic (1%)	Used a shallow copy or direct assignment, causing shared memory.	Attempted a deep copy but logic is flawed or incomplete.	**Successfully used Deep Copy** to ensure order2 has a unique product instance.
Sorting & Array Logic (1.5%)	Array not created or sorting logic is missing/broken.	Array created but sorting is incorrect or uses inefficient methods.	**Products correctly stored in an array and sorted by price** as required.
Object Creation & Testing (1%)	Testing class missing or objects not instantiated.	Testing class exists but objects are incomplete or unrelated.	Testing class properly instantiates objects and demonstrates all features.
Output & Summary (0.5%)	Output missing or incorrect.	Output is incomplete or difficult to interpret.	Output clearly displays details via printOrderDetails() or toString().