Coding Age

Practice Set : Java

---

1. Class Hierarchy and Inheritance in Employee Management System

Problem Statement:

Design a simple Employee Management System for a company using Object-Oriented Programming concepts. The system should consist of a base class and two derived classes that utilize inheritance and method overriding. Your goal is to manage employee information and calculate bonuses based on specific criteria.

Requirements:

1. **Base Class: Employee**
    - Create a class named Employee with the following attributes:
        - name (String): Represents the name of the employee.
        - id (Integer): A unique identifier for each employee.
        - salary (Double): The base salary of the employee.
    - Include an abstract method named calculateBonus() which will be implemented by the subclasses. This method will return the bonus amount for each employee.
2. **Subclass: Manager**
    - Create a subclass called Manager that inherits from Employee.
    - Add an additional attribute:
        - teamSize (Integer): Represents the number of team members managed by the manager.
    - Implement the calculateBonus() method to compute the bonus as follows:
        - The bonus will be 5% of the manager's salary for each team member.

        - Example: If the salary is $60,000 and the team size is 4, the bonus would be calculated as: Bonus=Salary×0.05×TeamSize\text{Bonus} = \text{Salary} \times 0.05 \times \text{TeamSize}Bonus=Salary×0.05×TeamSize
3. **Subclass: Developer**
    - Create another subclass called Developer that inherits from Employee.
    - Include an attribute:
        - skillLevel (String): This will represent the skill level of the developer, which can have values such as "junior", "mid", and "senior".
    - Implement the calculateBonus() method to determine the bonus percentage based on the skill level:
        - Junior: 5%
        - Mid: 10%
        - Senior: 15%
    - Example: If a senior developer has a salary of $80,000, the bonus would be: Bonus=Salary×0.15\text{Bonus} = \text{Salary} \times 0.15Bonus=Salary×0.15
4. **Department Class**
    - Create a Department class that maintains a list of employees.
    - Include the following methods:
        - **Add Employee**: Add an employee (of type Employee, Manager, or Developer) to the department.
        - **Remove Employee**: Remove an employee from the department based on their unique identifier.
        - **Calculate Total Bonus**: Compute the total bonus payout for all employees in the department. This method should iterate through the list of employees, calling calculateBonus() for each employee, and summing the results.

Objective:

This exercise will enhance your understanding of:

- **Inheritance**: Understanding how subclasses can inherit from a base class.
- **Method Overriding**: Implementing abstract methods in derived classes.
- **Encapsulation**: Managing the attributes and methods of your classes effectively.
- **Polymorphism**: Using the same method name (calculateBonus()) for different implementations in subclasses.

Example Usage:

- Create instances of Manager and Developer.
- Add these instances to a Department.
- Calculate and display the total bonuses for the department.

---

2. Encapsulation and Validation in a Student Grading System

Problem Statement:

Design a Student class that manages student information and handles their marks in various subjects. This system will emphasize encapsulation and validation to ensure data integrity.

Requirements:

1. **Class: Student**
    - Create a class named Student with the following private attributes:
        - name (String): Represents the name of the student.
        - rollNumber (Integer): A unique identifier for each student.
        - marks (Array of Integers): An array that holds the marks for different subjects (e.g., Math, Science, English).
2. **Methods:**
    - **setMarks(marks: Integer[])**:
        - Accept an array of marks as input.
        - Validate that each mark is within the range of 0 to 100.
        - If any mark is invalid (less than 0 or greater than 100), simply print a message indicating that the mark is invalid and do not update the marks array.

- **calculateAverage()**:
  - Calculate and return the average of the marks in the marks array.
  - If no marks have been set, return 0 or an appropriate message.
- **determineGrade()**:
  - Based on the calculated average, return the corresponding grade according to the following criteria:
    - A: 90 and above
    - B: 80–89
    - C: 70–79
    - D: 60–69
    - F: Below 60

Objective:

This exercise aims to enhance your understanding of:

- **Encapsulation**: Utilizing private attributes to safeguard class data.
- **Validation**: Implementing validation checks in setter methods to maintain data integrity.

Example Usage:

- Create an instance of the Student class.
- Use the setMarks() method to assign marks while validating them.
- Call calculateAverage() to compute the average of the marks.
- Use determineGrade() to get the grade based on the average marks.

---

3. Polymorphism in a Banking System with Account Types

Problem Statement:

Design a banking system that effectively manages various types of bank accounts. Utilize object-oriented programming principles, specifically **polymorphism**, **abstract classes**, and **interfaces**. Your system should support

the functionality of deposits and withdrawals, with specific rules for each account type.

Requirements:

1. **Abstract Class: Account**
    - Create an abstract class named Account with the following abstract methods:
        - **deposit(amount: Double)**: This method will be used to deposit money into the account.
        - **withdraw(amount: Double)**: This method will be used to withdraw money from the account.
2. **Subclass: SavingsAccount**
    - Extend the Account class to create a subclass named SavingsAccount.
    - Add the following attribute:
        - **maxWithdrawalLimit (Double)**: This attribute defines the maximum amount that can be withdrawn from the account in a month.
    - Implement the withdraw() method:
        - Ensure that the total amount withdrawn in a month does not exceed the maxWithdrawalLimit. If the withdrawal exceeds this limit, display a message indicating that the withdrawal cannot be processed due to exceeding the limit.
3. **Subclass: CurrentAccount**
    - Extend the Account class to create another subclass named CurrentAccount.
    - Add the following attribute:
        - **overdraftLimit (Double)**: This attribute allows the account to go negative up to a specified limit.
    - Implement the withdraw() method:
        - Allow withdrawals that exceed the current balance while keeping track of any overdraft fees. If the account balance goes negative, display a message indicating the remaining overdraft limit.
4. **Interface: InterestBearing**

    - Define an interface named InterestBearing with the following method:
        - **applyInterest()**: This method will be used to apply interest to the account balance.
    - Implement this interface in the SavingsAccount class to calculate and apply interest to the balance at the end of each month.

Objectives:

This exercise will help you:

- Understand and implement **polymorphism** through method overriding in subclasses.
- Utilize **abstract classes** to create a common interface for different account types, enforcing the implementation of specific methods.
- Use **interfaces** to define additional capabilities that are specific to certain subclasses.

Example Usage:

- Create instances of both SavingsAccount and CurrentAccount.
- Perform deposit and withdrawal operations on these account types, demonstrating their unique behaviors:
    - For SavingsAccount, show how it restricts withdrawals based on the monthly limit.
    - For CurrentAccount, illustrate how it allows withdrawals beyond the balance while tracking overdrafts.
- At the end of the month, apply interest to the SavingsAccount and observe how it affects the account balance.

---

4. Problem Statement: Company Hierarchy

Objective:
This exercise aims to enhance your understanding of multi-level inheritance and constructor chaining in Java. You will create a class hierarchy that models a company, including the classes Person, Employee, and Manager.

Class Structure

1. Person Class

- **Attributes:**
  - name (String): Represents the name of the person.
  - age (Integer): Represents the age of the person.
- **Constructor:**
  - Create a constructor that accepts the parameters name and age, initializing the respective attributes.

2. Employee Class

- **Inheritance:**
  - This class inherits from the Person class.
- **Attributes:**
  - salary (Double): Represents the salary of the employee.
  - department (String): Represents the department in which the employee works.
- **Constructor:**
  - Create a constructor that takes name, age, salary, and department as parameters. This constructor should use **constructor chaining** to call the superclass constructor, ensuring name and age are initialized correctly.

3. Manager Class

- **Inheritance:**
  - This class inherits from the Employee class.
- **Attributes:**
  - bonus (Double): Represents the bonus that the manager receives.
- **Constructor:**
  - Create a constructor that takes name, age, salary, department, and bonus as parameters. This constructor should use constructor chaining to initialize all attributes by invoking the appropriate superclass constructors.

Method Implementation

- **getTotalSalary():**
    - In the Manager class, implement a method called getTotalSalary(). This method should calculate and return the total salary by summing the salary and bonus attributes.

---

5. Problem Statement: Library System

Objective: This exercise is designed to help you simulate a real-world library scenario using Object-Oriented Programming (OOP) concepts. You will develop a library system that includes classes for Book, Library, and Member. The task will focus on understanding the relationships between these classes and how they manage their states.

Class Structure

1. **Book Class**
    - **Attributes:**
        - title (String): Represents the title of the book.
        - author (String): Represents the author of the book.
        - isbn (String): Represents the International Standard Book Number (ISBN) of the book.
        - availability (Boolean): Indicates whether the book is available for borrowing (true) or currently checked out (false).
    - **Constructor:**
        - Initializes title, author, isbn, and sets availability to true.
    - **Methods:**
        - setAvailability(boolean available): Updates the availability status of the book.
        - isAvailable(): Returns the current availability status of the book (true or false).
2. **Library Class**
    - **Attributes:**
        - books (List<Book>): A collection of all books available in the library.
    - **Constructor:**

        - Initializes the books collection.
    - **Methods:**
        - addBook(Book book): Adds a new book to the library's collection.
        - removeBook(Book book): Removes a specified book from the library's collection.
        - checkAvailability(Book book): Checks if the specified book is available for borrowing.
3. **Member Class**
    - **Attributes:**
        - name (String): Represents the name of the library member.
        - borrowedBooks (List<Book>): A collection of books currently borrowed by the member.
    - **Constructor:**
        - Initializes the member's name and the borrowedBooks collection.
    - **Methods:**
        - borrowBook(Book book): Marks the book as unavailable and adds it to the member's borrowedBooks collection.
        - returnBook(Book book): Marks the book as available and removes it from the member's borrowedBooks collection.

Objectives

- **Understand OOP Principles:** Gain experience with class creation, object instantiation, and method implementation.
- **Simulate Library Operations:** Model real-world activities like adding, removing, borrowing, and returning books.
- **Implement Relationships:** Learn how different classes interact and maintain state through methods and attributes.

Example Usage

Consider a library called "City Library":

1. **Adding Books:** The librarian creates Book objects, like "The Great Gatsby" by F. Scott Fitzgerald, and adds them to the Library using the addBook method.
2. **Checking Availability:** Member Alice wants to borrow "The Great Gatsby." The librarian uses checkAvailability to confirm it is available.
3. **Borrowing a Book:** Alice decides to borrow the book. The librarian calls borrowBook, marking it as unavailable and adding it to Alice's borrowedBooks list.
4. **Returning a Book:** After a week, Alice returns the book. The librarian calls returnBook, making it available again in the library.
5. **Managing Operations:** The librarian can add or remove books and track borrowed items, ensuring effective library management.

---

6. Problem Statement: Multiple Inheritance Using Interfaces in a Smart Device

Objective: This exercise aims to help you practice multiple inheritance through interfaces in Java. You will create a SmartDevice class that implements two interfaces, Playable and Recordable. This task will involve implementing logic that responds to the device's state realistically.

Class and Interface Structure

1. **Interfaces:**
    - **Playable Interface:**
        - **Method:**
            - play(): Starts playback of content.
    - **Recordable Interface:**
        - **Method:**
            - record(): Starts recording content.
2. **SmartDevice Class:**
    - **Attributes:**
        - deviceName (String): Represents the name of the smart device.

        - batteryLevel (Integer): Represents the battery percentage of the device.
        - storageCapacity (Integer): Represents the available storage space on the device.
    - **Methods:**
        - play(): Implements the playback functionality.
            - **Conditions:**
                - If batteryLevel is below 10%, display a warning message and prevent playback.
        - record(): Implements the recording functionality.
            - **Conditions:**
                - If batteryLevel is below 10%, display a warning message and prevent recording.
                - If storageCapacity is low, display an error message and prevent recording.

Example Scenario

- Create a SmartDevice object with a device name, battery level, and storage capacity.
- Call the play() method and observe how the device responds based on its battery level.
- Call the record() method and see how it behaves based on the battery level and storage capacity.

---

7. Problem Statement: Advanced Hotel Booking System with Custom Room Types

Objective:
Develop a hotel booking system that effectively utilizes design patterns—specifically the Factory and Builder patterns. Your system should manage different types of rooms and accommodate complex booking requests.

Class Structure

1. **Room Classes:**
    - **Base Room Class:** Create a basic Room class that contains common attributes shared by all room types.
    - **Specific Room Types:** Extend the Room class to define different types of rooms:
        - **Single Room:**
            - **Attributes:**
                - price (Double): Cost per night.
                - amenities (List<String>): Features available (e.g., Wi-Fi, TV).
        - **Double Room:**
            - **Attributes:**
                - price (Double): Cost per night.
                - amenities (List<String>): Features available.
        - **Suite Room:**
            - **Attributes:**
                - price (Double): Cost per night.
                - amenities (List<String>): Features available.
2. **Factory Pattern:**
    - **RoomFactory Class:** Implement a RoomFactory that creates room objects based on user preferences (e.g., room type and amenities).
    - The factory method should return the appropriate room type, streamlining the booking process.
3. **Builder Pattern:**
    - **Booking Class:** Design a Booking class that represents a booking request, allowing users to include optional features, such as:
        - **Breakfast:** Whether breakfast is included.
        - **Airport Shuttle:** Whether shuttle service is needed.
        - **Special Requests:** Any additional needs from the guest.
    - **BookingBuilder Class:** Create a BookingBuilder class that enables users to construct their booking step-by-step, making the process intuitive and flexible.
4. **BookingManager Class:**
    - **BookingManager Responsibilities:**

- **Checking Availability:** Determine which rooms are currently available.
- **Making Reservations:** Confirm bookings for guests.
- **Cancelling Bookings:** Handle cancellations and manage overbooked situations.

Example Scenario

Imagine a guest who wants to book a room at your hotel:

1. The guest selects a room type (Single, Double, or Suite) and specifies any additional features, such as breakfast or an airport shuttle.
2. The RoomFactory generates the appropriate room object based on their selections.
3. The guest uses the BookingBuilder to create their booking request, including all desired optional features.
4. The BookingManager checks room availability and processes the reservation, confirming the booking with all relevant details.

Expected Outcomes

By completing this project, you will:

- Gain practical experience in applying important design patterns in software development.
- Understand how to manage different classes and their interactions in a real-world scenario.
- Develop a functional hotel booking system that effectively handles customer requests and bookings.
