Encapsulation

1. Bank Account System with Interest Calculation and Transaction Fees

Requirements:

- Create a `BankAccount` class with private fields: `accountNumber`, `balance`, and `transactionHistory` (a list of strings). Use getter and setter methods for `balance` and `accountNumber`.
- Add a setter for `balance` that ensures the balance can never be negative.
- Add a method `deposit(double amount)` to add money, and `withdraw(double amount)` to subtract money (if balance is sufficient). Every withdrawal should incur a fixed fee (e.g., $2 fee).
- Create a method `calculateInterest(double interestRate)` that returns the interest for the current balance.
- Implement a method `getTransactionHistory()` to return the list of all transactions.
- In the `Main` class, simulate a series of deposits and withdrawals, and display the transaction history along with the final balance and interest.

---

2. Employee Salary and Bonus System with Complex Validation

Design an `Employee` class that manages an employee's salary and bonus. The employee's salary can change based on performance, and the bonus should be calculated based on their department's overall performance.

Requirements:

- Create a class `Employee` with private fields: `empId`, `name`, `baseSalary`, and `bonusPercentage`. Use setters with validation to ensure that no negative values are set for salary or bonus percentage.

- Create methods to compute:
  - `getGrossSalary()`: Calculates total salary based on base salary and bonus percentage.
  - `applyBonus(double departmentPerformance)`: Calculates bonus based on department performance. (e.g., 10% bonus if department performance is above 80%).
  - `updateSalary(double newBaseSalary)`: Updates the salary while ensuring it is a positive value.
- In the `Main` class, create multiple employee objects, set their salary details, and print out their updated salary after applying bonuses and validations.

---

3. Student Grade Management with Validation and GPA Calculation

Design a `Student` class where the grade input is encapsulated, and any grade outside the valid range (0–100) throws an exception. Calculate the student's GPA and handle the exception gracefully in the `Main` class.

Requirements:

- Create a class `Student` with private fields: `name`, `rollNumber`, and `grades[]` (array of integers). Create a setter method for grades that throws an exception if any grade is outside the valid range (0–100).
- Add a method `calculateGPA()` to compute the GPA as the average of the grades.
- Implement the logic to calculate GPA considering different weightings for courses, e.g., each course might contribute differently to the overall GPA.
- In the `Main` class, handle exceptions gracefully when setting invalid grades and compute the GPA for a student.

Inheritance

1. Multi-Level Employee System with Department and Role Hierarchy

Design a multi-level employee system where employees inherit common properties from a base class but also inherit additional responsibilities from their department-specific roles.

Requirements:

- Create a base class `Employee` with common attributes: `empId`, `name`, and `salary`. Define an abstract method `calculateSalary()`.
- Create a `Department` class with attributes like `departmentName`, and a method `addEmployee(Employee emp)` to assign employees to a department.
- Create subclasses of `Employee`:
  - `Manager`: Adds a `department` field and overrides `calculateSalary()` to include a departmental allowance.
  - `Engineer`: Adds a `project` field and overrides `calculateSalary()` to calculate salary based on project performance.
  - `SalesExecutive`: Calculates salary based on sales targets achieved.
- Implement a `Company` class that has a list of `Employee` objects and a method `calculateTotalSalaries()` to calculate total salary expenses for all employees in the company.

2. Online Course Management System with Different User Roles

Design a system where users (Students and Instructors) can enroll, create, or manage courses. Instructors can upload materials, while students can enroll and view course content.

Requirements:

- Create a base class `User` with fields like `String userId`, `String name`, and an abstract method `displayRole()`.

- Create subclasses:
    - `Instructor`: Implements `displayRole()` and adds methods to upload materials to courses.
    - `Student`: Implements `displayRole()` and adds methods to enroll in and view courses.

- Create a `Course` class with fields like `courseName`, `Instructor`, `List enrolledStudents`. Add methods to enroll students and assign instructors.

- Implement functionality in the `Main` class to simulate a student enrolling in a course, an instructor uploading materials, and printing student and instructor roles.

---

3. Vehicle System with Multiple Types and Custom Behaviors

Design a system to manage different vehicle types with varying behaviors like fuel consumption and maintenance.

Requirements:

- Create a base class `Vehicle` with fields like `model`, `fuelEfficiency`, `maxSpeed`, and an abstract method `displayDetails()`.

- Create subclasses:
    - `ElectricVehicle`: Adds `batteryLife` and overrides `displayDetails()` to include battery info.
    - `DieselVehicle`: Adds `fuelTankCapacity` and overrides `displayDetails()` to include fuel tank info.
    - `HybridVehicle`: Combines the behavior of both electric and diesel vehicles, overriding `displayDetails()` to provide detailed information for both.

- Create a `Fleet` class that holds a collection of vehicles and provides methods to add and remove vehicles. Add functionality to calculate total fuel efficiency for the fleet.

- Implement functionality to demonstrate polymorphism, where a list of vehicles is iterated, and the `displayDetails()` method is called dynamically.

Interfaces & Abstract Classes

1. Payment System with Multiple Payment Providers

Design a payment system where different payment providers (e.g., credit card, UPI, wallet) implement a common payment interface. Each payment provider has its own unique way of processing payments.

Requirements:

- Create an interface `PaymentMethod` with a method `processPayment(double `amount)`.

- Create an abstract class `OnlinePayment` that implements `PaymentMethod` with fields like `accountId`, and a concrete method `connect()` that establishes a connection to the payment gateway.

- Create subclasses:
  - `CreditCardPayment`: Implements `processPayment()` to handle payments via credit card, including credit card details and transaction fee.
  - `UPIPayment`: Implements `processPayment()` to handle UPI payments, including UPI ID and transaction limit.
  - `WalletPayment`: Implements `processPayment()` to handle payments through a digital wallet, including wallet balance check.

- In the `Main` class, demonstrate processing payments with multiple methods and handle transaction limits, fees, and success/failure statuses.

---

2. Smart Home System with Different Device Behaviors

Design a smart home management system where different types of smart appliances can be controlled via a common interface, with specific behaviors for each type of appliance.

Requirements:

- Create an interface `SmartDevice` with methods like `turnOn()`, `turnOff()`, and `getStatus()`.

- Create an abstract class `Appliance` that implements `SmartDevice` with a field `brand` and a method `connect()`.

- Create subclasses of `Appliance`:
  - `SmartLight`: Implements `turnOn()` and `adjustBrightness()` based on lighting preferences.
  - `SmartThermostat`: Implements `turnOn()` and `setTemperature()` to control the thermostat settings.
  - `SmartLock`: Implements `turnOn()` and `lock()`/`unlock()` to manage door locks.

- In the `Main` class, simulate controlling various devices in a smart home, turning devices on/off, adjusting settings, and printing status.

---

3. Employee Attendance and Leave System with Role-based Policies

Design a system to manage employee attendance, where different employee roles (e.g., Manager, Developer, Intern) have different leave policies. Track attendance and leave balance.

Requirements:

- Create an interface `Employee` with methods: `markAttendance()`, `getDetails()`, and `applyLeave(int days)`.

- Create an abstract class `FullTimeEmployee` that implements `Employee` and includes fields like `monthlySalary` and `leaveBalance`.

- Create subclasses:
  - `Manager`: Adds fields for managing teams and overrides `markAttendance()` to track manager-specific attendance.
  - `Developer`: Adds a field `project` and overrides `applyLeave()` to track leave days against project deadlines.

    - `Intern`: Tracks part-time leave and overrides `markAttendance()` based on hourly work.

- In the `Main` class, create instances of Manager, Developer, and Intern, and demonstrate tracking attendance, applying leave, and viewing leave balances.

Collections

1. Order Management System Using HashMap

Design an order management system where each customer can place multiple orders. Use a HashMap to store orders by customer ID, where each value is a list of orders.

Requirements:

- Create a class `Order` with fields: `orderId`, `productName`, `quantity`, and `price`.
- Create a `Customer` class with fields: `customerId`, name, and a `HashMap orders`

to store multiple orders.

- Implement methods in `Customer` to:
    - Add an order.
    - View all orders.
    - Calculate the total value of all orders for a customer.

- In the `Main` class, simulate adding orders for multiple customers, and display their order history along with the total value.

---

2. Hotel Booking System Using TreeSet

Design a hotel booking system where guests can book rooms. Use a `TreeSet` to manage available rooms in sorted order. The system should allow checking room availability, booking a room, and canceling a booking.

Requirements:

- Create a Room class with fields: `roomNumber`, `roomType`, `pricePerNight`, and implement `Comparable` to allow sorting based on `roomNumber`.
- Create a `Hotel` class with a `TreeSet` to store available rooms.
- Implement methods:
  - `bookRoom(Room room)`: Books a room if it is available.
  - `cancelBooking(Room room)`: Cancels a booking and makes the room available again.
  - `viewAvailableRooms()`: Displays all available rooms sorted by `roomNumber`.
- In the `Main` class, simulate booking and canceling rooms, and display the status of available rooms.

---

3. Inventory System Using LinkedHashMap

Design an inventory system to manage product stock using `LinkedHashMap` to maintain insertion order. Implement functionality to add, remove, and update products.

Requirements:

- Create a `Product` class with fields: `productId`, `productName`, `quantity`, and `price`.

- Create an `Inventory` class with a `LinkedHashMap` to store products by their `productName`.

- Implement methods to:
  - `addProduct(Product product)`: Adds a product to the inventory.
  - `removeProduct(String productName)`: Removes a product from the inventory.
  - `updateProductQuantity(String productName, int quantity)`: Updates the quantity of an existing product.
  - `getProductDetails(String productName)`: Returns details of a product by name.

- In the `Main` class, simulate adding, updating, and removing products, and display the inventory.
