ENCAPSULATION

QUESTION 1

You are tasked with creating a **CommissionEmployee** class to represent an employee who is paid based on a percentage of their gross sales. This class will enforce proper encapsulation and include validations for its fields.

CommissionEmployee Class:

Private fields:

- `firstName` (String): The employee's first name.
- `lastName` (String): The employee's last name.
- `socialSecurityNumber` (String): The employee's social security number.
- `grossSales` (double): The employee's total gross sales (must be greater than or equal to `0.0`).
- `commissionRate` (double): The percentage of gross sales paid as commission (must be between `0.0` and `1.0`).

Methods:

- Constructor to initialize all fields with proper validations.
- Getters and setters for all fields with the following validations:
  - `grossSales`: Must be greater than or equal to `0.0`. If not, throw an exception.
  - `commissionRate`: Must be between `0.0` and `1.0`. If not, throw an exception.
- `earnings()` - A method to calculate the employee's earnings based on the formula: `grossSales` * `commissionRate`

Task:

1. Create the `CommissionEmployee` class as specified above.
2. Write a **main method** to:
   - Create a `CommissionEmployee` object.
   - Display the employee's details using the `toString()` method.
   - Update the employee's `grossSales` and `commissionRate` and display the updated details.
   - Calculate and display the employee's earnings using the `earnings()` method.
   - Test the validation by trying to set invalid values for `grossSales` and `commissionRate`, and handle exceptions gracefully.

QUESTION 2

You are tasked with implementing a **Library Management System** using the principles of encapsulation. The system should manage books and library members. It should allow members to borrow books, return books, and track book availability.

Private fields

- `bookId` (String)
- `title` (String)
- `author` (String)
- `availableCopies` (int)

Encapsulated methods:

- Constructor to initialize a book with all attributes.
- Getters and setters for all fields.
- `borrowBook()` - Decrease the `availableCopies` by 1 if a copy is available, otherwise throw an exception.
- `returnBook()` - Increase the `availableCopies` by 1.

Member Class:

Private fields:

- `memberId` (String)
- `name` (String)
- `borrowedBooks` (ArrayList<Book>) - List of books borrowed by the member.

Encapsulated methods:

- Constructor to initialize a member with `memberId` and `name`.
- Getters and setters for all fields.
- `borrowBook(Book book)` - Allow the member to borrow a book if they haven't already borrowed it and the book is available.
- `returnBook(Book book)` - Allow the member to return a borrowed book.

Library Class:

Private fields:

- `books` (ArrayList<Book>) - List of all books in the library.
- `members` (ArrayList<Member>) - List of all members of the library.

Encapsulated methods:

- Add books to the library.
- Register new members.
- Search for a book by title or author.
- Check if a book is available.
- Allow a member to borrow or return a book.
- Display all books and their availability.

Task:

1. Implement the classes with encapsulation.
2. Write a **main method** that:
   - Creates a library with at least 5 books and 3 members.
   - Simulates borrowing and returning books.

QUESTION 3

You are tasked with designing a **Hospital Management System** that uses encapsulation to securely manage patient and doctor details.

Class: Patient:

- Private fields:
  - `patientId` (String): A unique identifier for the patient.
  - `name` (String): The name of the patient.
  - `age` (int): The age of the patient.
  - `diagnosis` (String): The current diagnosis of the patient.

    -
- Constructor:
    - A four-argument constructor to initialize `patientId`, `name`, `age`, and `diagnosis`.
- Methods:
    - Getters and setters for all fields:
        - `getPatientId()`, `getName()`, `getAge()`, `getDiagnosis()`.
        - `setAge(int age)`:
            - Ensures that `age` is greater than 0.
            - Prints: `"Invalid age"` if the validation fails.
        - `setDiagnosis(String diagnosis)`:
            - Ensures that `diagnosis` is not empty or null.
            - Prints: `"Diagnosis cannot be empty"` if the validation fails.
    - `updateDiagnosis(String newDiagnosis)`:
        - Updates the `diagnosis` field with the new value.
        - Prints: `"Diagnosis updated successfully to: <newDiagnosis>"`

Class: Doctor:

- Private fields:
    - `doctorId` (String): A unique identifier for the doctor.
    - `name` (String): The name of the doctor.
    - `specialization` (String): The field of specialization for the doctor.
    - `patientsTreated` (int): The number of patients treated by the doctor.
- Constructor:
    - A three-argument constructor to initialize `doctorId`, `name`, and `specialization`.
- Methods:
    - Getters and setters for all fields:
        - `getDoctorId()`, `getName()`, `getSpecialization()`, `getPatientsTreated()`.
    - `treatPatient()`:
        - Increments `patientsTreated` by 1 and prints: `"Patient treated successfully. Total patients treated: <patientsTreated>"`.

Tasks:

1. Implement the `Patient` and `Doctor` classes as described above.
2. In the main method:
    - Create a `Patient` object with the following details:
        - Patient ID: "P001"
        - Name: "John Smith"
        - Age: 45
        - Diagnosis: "Fever"
    - Create a `Doctor` object with the following details:
        - Doctor ID: "D101"
        - Name: "Dr. Alice"
        - Specialization: "General Medicine"
    - Perform the following operations:
        - Update the patient's diagnosis to "Flu".
        - Treat the patient and increase the doctor's `patientsTreated` count.
        - Attempt to set the patient's age to a negative value.
        - Attempt to update the patient's diagnosis to an empty string.

QUESTION 4

You are tasked with designing an **Airline Reservation System** that uses encapsulation to securely manage flight details, passenger information, and reservation operations.

Class: Flight:

- Private fields:
    - `flightNumber` (String): A unique identifier for the flight.
    - `destination` (String): The flight's destination.
    - `capacity` (int): The total number of seats on the flight.
    - `bookedSeats` (int): The number of seats currently booked.
- Constructor:
    - A four-argument constructor to initialize `flightNumber`, `destination`, `capacity`, and `bookedSeats`.
- Methods:
    - Getters and setters for all fields:
        - `getFlightNumber()`, `getDestination()`, `getCapacity()`, `getBookedSeats()`.

        - `setCapacity(int capacity)`:
            - Ensures that `capacity` is greater than or equal to `bookedSeats`.
            - Prints: `"Invalid capacity. It must be greater than or equal to booked seats."` if validation fails.
    - `bookSeat()`:
        - Increments `bookedSeats` by 1 if `bookedSeats < capacity`.
        - Prints: `"Seat booked successfully. Remaining seats: <capacity - bookedSeats>"` or `"No seats available."`
    - `cancelSeat()`:
        - Decrements `bookedSeats` by 1 if `bookedSeats > 0`.
        - Prints: `"Seat cancellation successful. Available seats: <capacity - bookedSeats>"` or `"No bookings to cancel."`

Class: Passenger:

- Private fields:
    - `passengerId` (String): A unique identifier for the passenger.
    - `name` (String): The name of the passenger.
    - `contactNumber` (String): The passenger's contact number.
    - `flightBooked` (String): The flight number of the flight the passenger is booked on (initially null).
- Constructor:
    - A three-argument constructor to initialize `passengerId`, `name`, and `contactNumber`.
- Methods:
    - Getters and setters for all fields:
        - `getPassengerId()`, `getName()`, `getContactNumber()`, `getFlightBooked()`.
        - `setContactNumber(String contactNumber)`:
            - Ensures that `contactNumber` is exactly 10 digits long.
            - Prints: `"Invalid contact number. It must be 10 digits."` if validation fails.
    - `bookFlight(String flightNumber)`:
        - Sets `flightBooked` to `flightNumber` and prints: `"Passenger booked on flight <flightNumber>."`
        - Does not allow booking if `flightBooked` is already set.
    - `cancelFlight()`:
        - Sets `flightBooked` to null if the passenger has an existing booking.

        - Prints: `"Flight booking cancelled."` or `"No booking found to cancel."`

Tasks:

1. Create a `Flight` object with the following details:
    - Flight Number: "AI101"
    - Destination: "New York"
    - Capacity: 200
    - Booked Seats: 150
2. Create a `Passenger` object with the following details:
    - Passenger ID: "P123"
    - Name: "Sarah Connor"
    - Contact Number: "9876543210"
3. Perform the following operations:
    - Book a seat on the flight for the passenger and update the flight details.
    - Attempt to book a flight for the same passenger again.
    - Cancel the passenger's flight booking and update the flight details.
    - Attempt to cancel a flight for the passenger when they do not have a booking.
    - Attempt to set an invalid capacity for the flight.
    - Attempt to set an invalid contact number for the passenger.

QUESTION 5

You are tasked with designing a **Banking System** that securely manages customer accounts, transactions, and financial analytics using advanced encapsulation principles.

Class: Account:

- Private fields:
    - `accountNumber` (String): A unique identifier for the account.
    - `accountHolderName` (String): Name of the account holder.
    - `balance` (double): The current balance in the account.
    - `accountType` (String): Type of account (e.g., "Savings", "Current").
- Constructor:
    - A four-argument constructor to initialize `accountNumber`, `accountHolderName`, `balance`, and `accountType`.

- Methods:
    - Getters and setters for all fields:
        - `getAccountNumber()`, `getAccountHolderName()`, `getBalance()`, `getAccountType()`.
        - `setAccountType(String accountType)`:
            - Ensures `accountType` is either "Savings" or "Current".
            - Prints: `"Invalid account type. Must be 'Savings' or 'Current'."` if validation fails.
    - `deposit(double amount)`:
        - Adds `amount` to `balance` if `amount > 0`.
        - Prints: `"Deposit successful. New balance: <balance>"` or `"Invalid deposit amount."`
    - `withdraw(double amount)`:
        - Deducts `amount` from `balance` if `amount <= balance`.
        - Prints: `"Withdrawal successful. Remaining balance: <balance>"` or `"Insufficient balance or invalid withdrawal amount."`

Class: Transaction:

- Private fields:
    - `transactionId` (String): A unique identifier for the transaction.
    - `accountNumber` (String): The account associated with the transaction.
    - `transactionType` (String): Type of transaction (e.g., "Deposit", "Withdrawal").
    - `amount` (double): The transaction amount.
    - `timestamp` (String): The date and time of the transaction.
- Constructor:
    - A five-argument constructor to initialize all fields.
- Methods:
    - Getters for all fields:
        - `getTransactionId()`, `getAccountNumber()`, `getTransactionType()`, `getAmount()`, `getTimestamp()`.
    - No setters: Transaction details should not be modifiable after creation.

Class: Bank:

- Private fields:
  - `accounts` (List<Account>): A list of all accounts in the bank.
  - `transactions` (List<Transaction>): A list of all transactions.
- Constructor:
  - Initializes `accounts` and `transactions` as empty lists.
- Methods:
  - `createAccount(String accountNumber, String accountHolderName, double initialDeposit, String accountType)`:
    - Adds a new `Account` object to `accounts` if the `accountNumber` is unique.
    - Prints: `"Account created successfully."` or `"Account number already exists."`
  - `processTransaction(String accountNumber, String transactionType, double amount)`:
    - Validates the transaction:
      - If `transactionType` is "Deposit", calls `deposit()` on the account.
      - If `transactionType` is "Withdrawal", calls `withdraw()` on the account.
      - Records a new `Transaction` object in `transactions`.
    - Prints: `"Transaction successful."` or appropriate error messages from the `Account` class.
  - `getAccountSummary(String accountNumber)`:
    - Prints account details and all transactions for the given `accountNumber`.
    - Prints: `"Account not found."` if the account does not exist.
  - `getBankAnalytics()`:
    - Calculates and prints:
      - Total accounts.
      - Total balance across all accounts.
      - Number of transactions and their breakdown by type (e.g., "Deposit", "Withdrawal").

Tasks:

1. Create a `Bank` object.
2. Add a method `getTopAccounts(int n)` in the `Bank` class to retrieve and print the top `n` accounts with the highest balance
3. Perform the following operations:
    - Create three accounts with unique account numbers:
        - Account 1: "A101", "Alice Johnson", 5000.0, "Savings"
        - Account 2: "A102", "Bob Smith", 2000.0, "Current"
        - Account 3: "A103", "Charlie Brown", 3000.0, "Savings"
    - Process the following transactions:
        - Deposit 1000.0 into Account 1.
        - Withdraw 2500.0 from Account 2.
        - Withdraw 4000.0 from Account 3 (should fail due to insufficient balance).
    - Retrieve and print the summary of Account 1, including transactions.
    - Print the overall bank analytics.

INHERITANCE

QUESTION 1

Extend the `CommissionEmployee` class from the previous exercise to a subclass `BasePlusCommissionEmployee`.

CommissionEmployee Class (Superclass):

Private Fields:

- `firstName: String`
- `lastName: String`
- `socialSecurityNumber: String`
- `grossSales: double`
- `commissionRate: double`

Constructor:

- Accept `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` as parameters and initialize the fields.
- Validate that `grossSales >= 0` and `0 < commissionRate < 1`.

Methods:

- `getFirstName()`: Returns the first name.
- `getLastName()`: Returns the last name.
- `getSocialSecurityNumber()`: Returns SSN.
- `getGrossSales()`: Returns gross sales.
- `getCommissionRate()`: Returns commission rate.
- `earnings()`: Returns `grossSales * commissionRate`.
- `toString()`: Returns a string with employee's details.

BasePlusCommissionEmployee Class (Subclass):

Private Fields:

- `baseSalary: double`

Methods**:**

- **`getBaseSalary()`:** Returns the base salary.
- **`setBaseSalary(baseSalary)`:** Sets the base salary and validates that it is greater than **`0.0`**.
- **`earnings()`:** Returns the sum of **`baseSalary`** and the earnings from the superclass (**`super.earnings()`**).
- **`toString()`**: Returns a string with employee's details including the base salary and earnings.

Constructor:

- Accept **`firstName`**, **`lastName`**, **`socialSecurityNumber`**, **`grossSales`**, **`commissionRate`**, **`baseSalary`** as parameters**.**
- Call the superclass constructor to initialize inherited fields.
- Set **`baseSalary`** using a setter.

QUESTION 2

You are tasked with developing a **Vehicle Rental Management System** using **inheritance** in Java. The system should manage different types of vehicles available for rent, calculate rental charges based on the vehicle type, and maintain the rental status of each vehicle.

Vehicle Class (Base Class)

Protected fields:

- `vehicleId` (String)
- `brand` (String)
- `model` (String)
- `isAvailable` (boolean)

Encapsulated methods:

- Constructor to initialize the vehicle with all attributes.
- Getters for all fields.

- `rentVehicle()` - Marks the vehicle as unavailable if it is available, otherwise throws an exception.
- `returnVehicle()` - Marks the vehicle as available.
- `calculateRentalCost(int days)` - An abstract method to calculate rental cost (to be implemented by derived classes).

Car Class (Derived Class)

Private fields:

- `seatingCapacity` (int)

Additional methods:

- Constructor to initialize a car with all attributes.
- Override `calculateRentalCost(int days)`:
    - Formula: `1000 * days` + `seatingCapacity * 50`.

Bike Class (Derived from Vehicle):

- Private fields:
    - `engineCapacity` (int) (in cc)
- Additional methods:
    - Constructor to initialize a bike with all attributes.
    - Override `calculateRentalCost(int days)`:
        - Formula: `500 * days` + `engineCapacity / 10`.

RentalManager Class:

- Maintains a list of vehicles (cars and bikes).
- Methods:
    - Add vehicles to the system.
    - Display all available vehicles.
    - Rent a vehicle by ID.
    - Return a vehicle by ID.

Task:

- Implement the classes with proper inheritance.
- Write a main method that:
    - Creates a fleet of vehicles (both cars and bikes).
    - Simulates renting and returning vehicles.
    - Displays the total rental cost based on the number of days.

QUESTION 3

You are tasked with designing an **E-Commerce System** to manage different types of users and orders using inheritance principles.

Class: User

- Fields:
  - `userId` (String): Unique identifier for the user.
  - `name` (String): Name of the user.
- Constructor:
  - A two-argument constructor to initialize `userId` and `name`.
- Methods:
  - `printUserDetails()`: prints `userId` and `name`

Class: Customer (Subclass of User)

- Fields:
  - `email` (String): Email address of the customer.
  - `cart` (List<String>): List of items currently in the cart.
- Constructor:
  - A three-argument constructor to initialize `userId`, `name`, and `email`.
  - Sets `cart` as an empty list.
- Methods:
  - `addItemToCart(String item)`:
    - Adds `item` to the `cart`.
    - Prints: `"Item '<item>' added to cart."`
  - `viewCart()`: Prints `'Cart for Customer <name>'` followed by numbered item list

Class: Admin (Subclass of User)

- Fields:
  - `permissions` (List<String>): A list of admin permissions (e.g., "Manage Orders", "View Reports").
- Constructor:
  - A three-argument constructor to initialize `userId`, `name`, and `permissions`.

- Methods:
    - `addPermission(String permission)`:
        - Adds a permission to `permissions`.
        - Prints: `"Permission '<permission>' added for Admin <name>."`
    - `viewPermissions()`: Prints `'Perimissions for Admin <name>'` followed by numbered Permissions

Class: Order

- Fields:
    - `orderId` (String): Unique identifier for the order.
    - `userId` (String): The ID of the user who placed the order.
    - `orderDetails` (List<String>): List of items in the order.
- Constructor:
    - A three-argument constructor to initialize `orderId`, `userId`, and `orderDetails`.
- Methods:
    - `printOrderDetails()`:
      Prints
      `'Order ID: <orderId>`
      `Placed by user ID: <userID>`
      `Items:`
      `1.<item1>`
      `2.<item2>'` etc

Main Class: ECommerceSystem

- Fields:
    - `users` (List<User>): List of all users in the system.
    - `orders` (List<Order>): List of all orders placed in the system.
- Constructor:
    - Initializes `users` and `orders` as empty lists.
- Methods:
    - `registerUser(User user)`:
        - Adds a new `User` to the `users` list.
        - Prints: `"User '<name>' registered."`
    - `placeOrder(String userId, List<String> items)`:
        - Creates a new `Order` if `userId` exists in `users`.
        - Adds the order to `orders`.

        - Prints: `"Order '<orderId>' placed successfully by User '<userId>'."`
    - `viewAllUsers()`:
        - Prints details of all users using `printUserDetails()`.
    - `viewAllOrders()`:
        - Prints details of all orders using `printOrderDetails()`.

Tasks:

1. Create an `ECommerceSystem` object.
2. Perform the following operations:
    - Register two customers and one admin:
        - Customer 1: `"C001"`, `"Alice"`, `"alice@example.com"`
        - Customer 2: `"C002"`, `"Bob"`, `"bob@example.com"`
        - Admin: `"A001"`, `"Eve"`, with permissions `"Manage Orders"`, `"View Reports"`
    - Customers add items to their cart:
        - Alice adds `"Laptop"` and `"Mouse"`.
        - Bob adds `"Smartphone"` and `"Headphones"`.
    - Place orders for the customers:
        - Alice places an order for the items in her cart.
        - Bob places an order for the items in his cart.
    - Admin adds a new permission `"Edit Products"`.
    - View all users and orders.

QUESTION 4

You are tasked with designing a **Hospital Management System** to manage different types of staff and their roles using inheritance principles.

Class: Staff

- Fields:
    - `staffId` (String): Unique identifier for the staff member.
    - `name` (String): Name of the staff member.
    - `department` (String): Department the staff belongs to.
- Constructor:
    - A three-argument constructor to initialize `staffId`, `name`, and `department`.

- Methods:
    - `displayDetails()`:
      Prints
      ```
      staff ID: <staffId>
       Name: <name>
       Department: <department>
      ```

Class: Doctor (Subclass of Staff)

- Fields:
    - `specialization` (String): Specialization of the doctor (e.g., "Cardiology", "Neurology").
    - `yearsOfExperience` (int): Number of years the doctor has been practicing.
- Constructor:
    - A five-argument constructor to initialize `staffId`, `name`, `department`, `specialization`, and `yearsOfExperience`.
- Methods:
    - Override `displayDetails()`:
      Prints
      ```
      staff ID: <staffId>
      Name: <name>
      Department: <department>
      Specialization: <specialization>
      Experience: <yearsOfExxperience> years
      ```

Class: Nurse (Subclass of Staff)

- Fields:
    - `shift` (String): The nurse's shift timing (e.g., "Day", "Night").
    - `patientsAssigned` (int): Number of patients assigned to the nurse.
- Constructor:
    - A five-argument constructor to initialize `staffId`, `name`, `department`, `shift`, and `patientsAssigned`.
- Methods:
    - Override `displayDetails()`

Prints

```
staff ID: <staffId>
Name: <name>
Department: <department>
Shift: <shift>
Patients Assigned: <patientsAssigned>
```

Main Class: HospitalManagementSystem

- Fields:
  - `staffList` (List<Staff>): List of all registered staff members.
- Constructor:
  - Initializes `staffList` as an empty list.
- Methods:
  - `registerStaff(Staff staff)`:
    - Adds a `Staff` object to the `staffList`.
    - Prints: `"Staff '<name>' registered successfully."`
  - `displayAllStaff()`:
    - Iterates through the `staffList` and calls `displayDetails()` for each staff member.

Tasks:

1. Create a `HospitalManagementSystem` object.
2. Perform the following operations:
   - Register two doctors and one nurse:
     - Doctor 1: `"S001"`, `"Dr. Smith"`, `"Cardiology"`, `"Cardiology"`, `15`.
     - Doctor 2: `"S002"`, `"Dr. Lee"`, `"Neurology"`, `"Neurology"`, `8`.
     - Nurse: `"S003"`, `"Nurse Kelly"`, `"Emergency"`, `"Night"`, `5`.
   - Display all registered staff.

QUESTION 5

You are tasked with designing a **Restaurant Management System** to manage various staff roles using inheritance principles. This system will focus on role-specific responsibilities and task delegation.

Class: Employee

- Fields:
  - `employeeId` (String): Unique identifier for the employee.
  - `name` (String): Name of the employee.
- Constructor:
  - A two-argument constructor to initialize `employeeId` and `name`.
- Abstract Method:
  - `performDuty()`:
    - This abstract method defines a role-specific task each employee performs.

Class: Chef (Subclass of Employee)

- Fields:
  - `specialty` (String): The type of cuisine the chef specializes in (e.g., "Italian", "Indian").
- Constructor:
  - A three-argument constructor to initialize `employeeId`, `name`, and `specialty`.
- Implementation:
  - Implement the `performDuty()` method to print `Chef <name> is preparing <specialty> dishes.`

Class: Waiter (Subclass of Employee)

- Fields:
  - `assignedSection` (String): The section of the restaurant the waiter is responsible for (e.g., "Indoor", "Outdoor").
- Constructor:
  - A three-argument constructor to initialize `employeeId`, `name`, and `assignedSection`.
- Implementation:
  - Implement the `performDuty()` method to print `Waiter <name> is serving customers in the <assignedSection> section.`

Class: RestaurantManagementSystem

- Methods:
    - `assignDuty(Employee employee)`:
        - Accepts an `Employee` object and calls the `performDuty()` method to delegate the task.
        - Example of polymorphism: `Employee` objects can represent either `Chef` or `Waiter`.

Tasks:

1. Create at least one `Chef` and one `Waiter` object:
    - Chef: `"E001"`, `"Alice"`, `"Italian"`.
    - Waiter: `"E002"`, `"Bob"`, `"Outdoor"`.
2. Add these employees to an array or list.
3. Iterate through the list and call `assignDuty()` for each employee.

POLYMORPHISM

QUESTION 1
You are tasked with creating a **BasePlusCommissionEmployee** class that extends the **CommissionEmployee** class. This new class will represent an employee who is paid a base salary in addition to a commission based on gross sales. Proper encapsulation and validation must be implemented for the new and inherited fields.

BasePlusCommissionEmployee Class:

Inherits from the `CommissionEmployee` class.

Private field:

- `baseSalary` (double): The employee's fixed base salary (must be greater than `0.0`).

Constructors:

A five-argument constructor in the superclass (`CommissionEmployee`) to initialize the following fields:

- `firstName` (String)
- `lastName` (String)
- `socialSecurityNumber` (String)
- `grossSales` (double)
- `commissionRate` (double)

A six-argument constructor in the subclass to initialize:

- `baseSalary` (double): Initialize using validation.
- Inherited fields (`firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate`): Pass these to the superclass constructor using `super`.

Methods:

- Getter and setter methods for `baseSalary` with the following validation:
  - `baseSalary`: Must be greater than `0.0`. If not, throw an exception.
- Override the `earnings()` method to calculate the total earnings as: `baseSalary` + (`grossSales` * `commissionRate`)

Task:

1. Create the **`BasePlusCommissionEmployee`** class as specified.
2. Write a **main method** to:
   - Create a `BasePlusCommissionEmployee` object using the six-argument constructor.
   - Update the `baseSalary` and inherited fields (`grossSales` and `commissionRate`) and display the updated details.
   - Calculate and display the total earnings using the overridden `earnings()` method.
   - Test the validation by attempting to set invalid values for `baseSalary`, `grossSales`, and `commissionRate`, and handle exceptions appropriately.

QUESTION 2

You are tasked with designing a **Banking System** that demonstrates **both compile-time (method overloading)** and **run-time polymorphism (method overriding)**. The system should handle different types of accounts and operations.

Class: BankAccount:

- Private fields:
  - `accountHolderName` (String): The name of the account holder.
  - `accountNumber` (String): The unique account number.
  - `balance` (double): The current account balance.
- Constructor:
  - A three-argument constructor to initialize `accountHolderName`, `accountNumber`, and `balance`.
- Methods:
  - `deposit(double amount)`:
    - Increases the balance by the amount specified.
    - Prints: `"Deposit successful. New balance: <balance>"`
  - Overloaded `deposit(double amount, String transactionNote)`:
    - Increases the balance and prints the transaction note along with the updated balance.
  - `withdraw(double amount)`:
    - Decreases the balance by the specified amount if sufficient funds are available.
    - Prints: `"Withdrawal successful. New balance: <balance>"` or `"Insufficient funds."`

Subclass: SavingsAccount (extends BankAccount):

- Additional private field:
  - `interestRate` (double): Annual interest rate for the savings account.
- Constructor:
  - A four-argument constructor to initialize `accountHolderName`, `accountNumber`, `balance`, and `interestRate`.
- Methods:
  - Override the `withdraw(double amount)` method:
    - Ensures that the balance does not drop below a minimum threshold (e.g., $100).
    - Prints: `"Withdrawal successful. New balance: <balance>"` or `"Minimum balance requirement not met."`

Subclass: CurrentAccount (extends BankAccount):

- Additional private field:
  - `overdraftLimit` (double): The overdraft limit for the account.
- Constructor:
  - A four-argument constructor to initialize `accountHolderName`, `accountNumber`, `balance`, and `overdraftLimit`.
- Methods:
  - Override the `withdraw(double amount)` method:
    - Allows withdrawal even if it exceeds the balance, as long as it is within the overdraft limit.
    - Prints: `"Withdrawal successful. New balance: <balance>"` or `"Overdraft limit exceeded."`

Task:

1. Implement the `BankAccount`, `SavingsAccount`, and `CurrentAccount` classes as described above.
2. Add a method `calculateInterest()` in `SavingsAccount` to calculate and display annual interest.
3. Add a method `displayAccountDetails()` in `BankAccount` to print account details. Override it in subclasses to include additional fields like `interestRate` or `overdraftLimit`.
4. In the main method:

- Create a `SavingsAccount` with a balance of $500 and an interest rate of 3%.
- Create a `CurrentAccount` with a balance of $1000 and an overdraft limit of $500.
- Call `deposit` for both accounts using one argument and two arguments (method overloading).
- Call `withdraw` for both accounts with amounts that:
  - For `SavingsAccount`: Try to withdraw an amount leaving the balance below $100 to test the overridden method.
  - For `CurrentAccount`: Try to withdraw an amount that exceeds the overdraft limit to test the overridden method.

QUESTION 3

You are tasked with designing a **Transportation Management System** to handle various types of vehicles and their operations using polymorphism. The system must demonstrate both **runtime polymorphism** (method overriding) and **compile-time polymorphism** (method overloading).

Abstract Class: Vehicle

- Fields:
  - `vehicleId` (String): Unique identifier for the vehicle.
  - `model` (String): Model name of the vehicle.
  - `fuelLevel` (double): Current fuel level of the vehicle in liters.
- Constructor:
  - A three-argument constructor to initialize `vehicleId`, `model`, and `fuelLevel`.
- Methods:
  - `refuel(double liters)`:
    - Adds `liters` to `fuelLevel`.
      Prints:
      `Refueled <liters> liters. New fuel level: <fuelLevel> liters.`
  - Abstract Method:
    - `calculateRange()`:
      - Must implement vehicle-specific range calculation in subclasses.

Class: Car (Subclass of Vehicle)

- Fields:
  - `fuelEfficiency` (double): Fuel efficiency in kilometers per liter.
- Constructor:
  - A four-argument constructor to initialize `vehicleId`, `model`, `fuelLevel`, and `fuelEfficiency`.
- Methods:
  - Override `calculateRange()`:
    - Calculate the range as: `fuelLevel * fuelEfficiency`.
    - Prints:
      `Car <model> can travel <range> kilometers.`

Class: Truck (Subclass of Vehicle)

- Fields:
  - `cargoWeight` (double): Weight of the cargo in kilograms.
- Constructor:
  - A four-argument constructor to initialize `vehicleId`, `model`, `fuelLevel`, and `cargoWeight`.
- Methods:
  - Override `calculateRange()`:
    - Reduce the fuel efficiency based on cargo weight: `effectiveFuelEfficiency = baseFuelEfficiency - (cargoWeight / 1000.0)`
    - Calculate range as: `fuelLevel * effectiveFuelEfficiency`.
    - Prints: `Truck <model> with <cargoWeight> kg cargo can travel <range> kilometers.`
    - If cargo weight is too high, print: `Truck <model> cannot operate with cargo over the limit.`

Class: TransportationManager

- Methods:
  - `operateVehicle(Vehicle vehicle)`:
    - Calls the `calculateRange()` method on the given `Vehicle` object.
    - Demonstrates **runtime polymorphism** by working with both `Car` and `Truck`

Tasks:

1. Create an array or list of `Vehicle` objects:
   - Car: `"C001"`, `"Sedan"`, `50.0`, `15.0` (fuel efficiency in km/l).
   - Truck: `"T001"`, `"Freightliner"`, `100.0`, `2000.0` (cargo weight in kg).
   - Truck: `"T002"`, `"Heavy Hauler"`, `80.0`, `10000.0` (cargo weight in kg).
2. Perform the following operations:
   - Refuel the Sedan with 10 liters and calculate its range.
   - Calculate the range for the Freightliner with its current cargo.
   - Calculate the range for the Heavy Hauler with its current cargo, considering that it exceeds operational limits.
3. Use the `operateVehicle()` method to handle these operations for all vehicle types.

QUESTION 4

You are tasked with designing an **E-Commerce System** to handle various types of products and dynamic pricing using polymorphism. The system must incorporate **runtime polymorphism** (method overriding) and **compile-time polymorphism** (method overloading).

Abstract Class: Product

- Fields:
  - `productId` (String): Unique identifier for the product.
  - `productName` (String): Name of the product.
  - `basePrice` (double): The base price of the product.
- Constructor:
  - A three-argument constructor to initialize `productId`, `productName`, and `basePrice`.
- Methods:
  - `applyDiscount(double percentage)`:
    - Reduces the `basePrice` by the given `percentage`.
      Prints:
      `Discount applied. New price: <basePrice>.`
  - Abstract Method:
    - `calculateFinalPrice()`:
      - Must calculate the final price based on product-specific rules in subclasses.

Class: Electronics (Subclass of Product)

- Fields:
  - `warrantyPeriod` (int): Warranty period in months.
- Constructor:
  - A four-argument constructor to initialize `productId`, `productName`, `basePrice`, and `warrantyPeriod`.
- Methods:
  - Override `calculateFinalPrice()`:
    - Add a `10% warranty fee` to the base price.
      Prints:
      `Final price of <productName> with warranty: <finalPrice>.`

Class: Clothing (Subclass of Product)

- Fields:
  - `size` (String): Size of the clothing item (e.g., "S", "M", "L").
  - `seasonalDiscount` (double): Seasonal discount in percentage.
- Constructor:
  - A five-argument constructor to initialize `productId`, `productName`, `basePrice`, `size`, and `seasonalDiscount`.
- Methods:
  - Override `calculateFinalPrice()`:
    - Subtract the `seasonalDiscount` from the base price.
      Prints:
      `Final price of <productName> after seasonal discount: <finalPrice>.`

Class: Cart

- Methods:
  - `addProduct(Product product)`:
    - Adds the product to the cart.
  - `calculateTotalPrice(Product... products)`:
    - Demonstrates **method overloading** by calculating the total price for multiple products passed as a variable-length argument.
    - Prints:
      plaintext
      Copy code
      `Total price of cart: <totalPrice>.`

Tasks:

1. Create an array or list of `Product` objects:
   - Electronics: `"E001"`, `"Laptop"`, `1000.0`, `24`.
   - Clothing: `"C001"`, `"Winter Jacket"`, `200.0`, `"M"`, `20.0`.
2. Perform the following operations:
   - Apply a `10%` discount to the `Laptop`.
   - Calculate the final price of the `Laptop` and `Winter Jacket`.
   - Add both products to the cart and calculate the total price using `Cart`.

QUESTION 5

You are tasked with designing a **Staff Management System** for an organization. The system must demonstrate **polymorphism** to handle different types of workers, including **method overriding** (runtime polymorphism) and **dynamic method dispatch**.

Base Class: StaffMember

- Fields:
  - `name` (String): The name of the staff member.
  - `id` (String): Unique identifier for the staff member.
- Constructor:
  - A two-argument constructor to initialize `name` and `id`.
- Methods:
  - `getPay()`:
    - Abstract method to calculate the monthly payment for a staff member.
    - Implementations of this method will vary across subclasses.
  - `toString()`:
    - Returns a description of the staff member, including their `name` and `id`.

Subclass: Employee (Derived from StaffMember)

- Fields:
  - `monthlySalary` (double): Fixed monthly salary for regular employees.
- Constructor:
  - A three-argument constructor to initialize `name`, `id`, and `monthlySalary`.
- Methods:
  - Override `getPay()`:
    - Returns `monthlySalary`.

Subclass: TemporaryEmployee (Derived from StaffMember)

  - Fields:
    - `hourlyRate` (double): Rate per hour worked.
    - `hoursWorked` (int): Total hours worked in a month.
  - Constructor:

        - A four-argument constructor to initialize `name`, `id`, `hourlyRate`, and `hoursWorked`.
    - Methods:
        - Override `getPay()`:
            - Returns `hourlyRate * hoursWorked`.

Subclass: Manager (Derived from Employee)

- Fields:
    - `bonus` (double): Monthly bonus amount.
- Constructor:
    - A four-argument constructor to initialize `name`, `id`, `monthlySalary`, and `bonus`.
- Methods:
    - Override `getPay()`:
        - Returns `monthlySalary + bonus`.

Subclass: Volunteer (Derived from StaffMember)

- Constructor:
    - A two-argument constructor to initialize `name` and `id`.
- Methods:
    - Override `getPay()`:
        - Returns `0.0` as volunteers are unpaid.

Class: Staff

- Fields:
    - `staffList` (Array of StaffMember): Stores all the workers in the organization.
- Methods:
    - `getTotalCost()`:
        - Loops through `staffList` and calls the `getPay()` method for each `StaffMember` object.
        - Returns the total payment required for all workers at the end of the month.
    - `displayStaff()`:
        - Loops through `staffList` and prints the details of each `StaffMember`.

Tasks:

1. Create an array or list of `StaffMember` objects:
    - Regular Employee: `"John"`, `"E001"`, `3000.0`.
    - Temporary Employee: `"Jane"`, `"E002"`, `15.0` (hourly rate), `160` (hours worked).
    - Manager: `"Alice"`, `"M001"`, `5000.0` (monthly salary), `2000.0` (bonus).
    - Volunteer: `"Mark"`, `"V001"`.
2. Implement the `getTotalCost()` method to calculate the total monthly payment for all workers.
3. Implement the `displayStaff()` method to print details of all workers and their pay.

ABSTRACTION

QUESTION 1

You are tasked with creating an **Employee** abstract class and a **FullTimeEmployee** subclass. The **Employee** class will serve as a base class for different types of employees. The **FullTimeEmployee** class will implement specific behavior for full-time employees.

Employee Class:

Private fields:

- `name` (String): The name of the employee.
- `employeeId` (String): The unique ID for the employee.

Constructor:

- A two-argument constructor to initialize the `name` and `employeeId` fields.

Methods:

- Getter methods for `name` and `employeeId`.
- An abstract method `calculatePay()` with no parameters.

FullTimeEmployee Class:

Inherits from the abstract `Employee` class.

Private field:

- `salary` (double): The full-time employee's salary.

Constructor:

- A three-argument constructor to initialize:
- `name` (String): Passed to the superclass constructor.
- `employeeId` (String): Passed to the superclass constructor.
- `salary` (double): Initialized in the subclass constructor.

Methods:

Override the `calculatePay()` method to print: `FullTimeEmployee Pay: <salary>`
A getter method for `salary`.

Task:

1. Create the abstract **`Employee`** class as specified.
2. Create the **`FullTimeEmployee`** subclass and implement the required functionality.
3. Write a **main method** to:
    - Create a `FullTimeEmployee` object using the three-argument constructor.
    - Display the employee's name, ID, and salary using the getter methods.
    - Call the `calculatePay()` method to display the pay.
    - Test the `calculatePay()` method for different salary values.

QUESTION 2

You are tasked with designing a **Medical Record Management System** to represent different types of medical personnel and their responsibilities. The system should use abstraction to enforce a common structure across all types of personnel while allowing specific details to vary.

Abstract Class: MedicalPersonnel:

Private fields:

- `name` (String): The name of the personnel.
- `id` (String): A unique ID for the personnel.

Constructor:

- A two-argument constructor to initialize `name` and `id`.

Methods:

- Getter methods for `name` and `id`.
- Abstract methods:
    - `performDuties()`:

        - Defines the duties performed by the personnel.
    - `getSpecialization()`:
        - Returns a string describing the specialization of the personnel.

- Concrete method:
    - `displayDetails()`:
    - Prints the `name` and `id` of the personnel.

Class: Doctor (Subclass of MedicalPersonnel):

- Additional private field:
    - `specialization` (String): The medical specialty (e.g., Cardiologist, Pediatrician).
- Constructor:
    - A three-argument constructor to initialize `name`, `id`, and `specialization`.
- Methods:
    - Implement the `performDuties()` method to print

      Doctor <name>: Diagnoses patients, prescribes medication, and conducts surgeries.

    - Implement the `getSpecialization()` method to return the `specialization` field.

Class: Nurse (Subclass of MedicalPersonnel):

- Additional private field:
    - `department` (String): The department the nurse works in (e.g., ICU, Emergency).
- Constructor:
    - A three-argument constructor to initialize `name`, `id`, and `department`.
- Methods:
    - Implement the `performDuties()` method to print

      Nurse <name>: Provides patient care, administers medications, and assists doctors.

    - Implement the `getSpecialization()` method to return the `department` field.

Class: Pharmacist (Subclass of MedicalPersonnel):

- Additional private field:
  - `pharmacyLicenseId` (String): The license ID of the pharmacist.
- Constructor:
  - A three-argument constructor to initialize `name`, `id`, and `pharmacyLicenseId`.
- Methods:
  - Implement the `performDuties()` method to print

    Pharmacist <name>: Dispenses medication and advises patients on drug usage.

  - Implement the `getSpecialization()` method to return `"Pharmacy"`.

Task:

1. Create the `MedicalPersonnel`, `Doctor`, `Nurse`, and `Pharmacist` classes as described above.
2. Write a **main method** to:
   - Create an array or list of `MedicalPersonnel` objects containing at least one `Doctor`, one `Nurse`, and one `Pharmacist`.
   - Use a loop to:
     - Call `displayDetails()` for each object to display its name and ID.
     - Call `performDuties()` for each object to display their responsibilities.
     - Call `getSpecialization()` to display their specialization or department.

QUESTION 3

You are tasked with designing a **Device Management System** for a tech company that manages different types of devices used by employees. The system must use **abstraction** to provide a blueprint for handling various device operations while allowing specific implementations for different device types.

Abstract Class: Device

- Fields:
  - `deviceId` (String): A unique identifier for the device.
  - `brand` (String): The brand of the device.
  - `model` (String): The specific model of the device.
- Constructor:
  - A three-argument constructor to initialize `deviceId`, `brand`, and `model`.

- Abstract Methods:
    - `calculatePowerConsumption()`:
        - Computes the power consumption of the device in kilowatt-hours.
    - `calculateMaintenanceCost()`:
        - Computes the yearly maintenance cost of the device.
- Concrete Methods:
    - `getDetails()`:
        - Returns a string containing the `deviceId`, `brand`, and `model`.

Concrete Class: Laptop (Derived from Device)

- Fields:
    - `processorPower` (double): Power of the processor in watts.
    - `dailyUsageHours` (double): Average hours the device is used daily.
    - `maintenanceCostPerYear` (double): Fixed yearly maintenance cost.
- Constructor:
    - A five-argument constructor to initialize all fields, including those from the `Device` class.
- Implementations:
    - `calculatePowerConsumption()`:
        - Returns `(processorPower * dailyUsageHours * 365) / 1000` (kWh per year).
    - `calculateMaintenanceCost()`:
        - Returns `maintenanceCostPerYear`

Concrete Class: Smartphone (Derived from Device)

- Fields:
    - `batteryCapacity` (double): Battery capacity in mAh.
    - `chargeCyclesPerYear` (int): Number of times the device is fully charged in a year.
    - `maintenanceCostPerCycle` (double): Maintenance cost per charge cycle.
- Constructor:
    - A five-argument constructor to initialize all fields, including those from the `Device` class.
- Implementations:

        - `calculatePowerConsumption()`:
            - Returns `(batteryCapacity * chargeCyclesPerYear * 3.7 / 1000)` (kWh per year, assuming 3.7V battery voltage).
        - `calculateMaintenanceCost()`:
            - Returns `(chargeCyclesPerYear * maintenanceCostPerCycle)`.

Concrete Class: Desktop (Derived from Device)

- Fields:
    - `powerConsumptionPerHour` (double): Power consumed per hour in watts.
    - `dailyUsageHours` (double): Average hours the device is used daily.
    - `yearlyServiceCost` (double): Fixed yearly service cost.
- Constructor:
    - A five-argument constructor to initialize all fields, including those from the `Device` class.
- Implementations:
    - `calculatePowerConsumption()`:
        - Returns `(powerConsumptionPerHour * dailyUsageHours * 365) / 1000` (kWh per year).
    - `calculateMaintenanceCost()`:
        - Returns `yearlyServiceCost`

Class: DeviceManager

- Fields:
    - `deviceList` (List of Device): Stores all types of devices.
- Methods:
    - `addDevice(Device device)`:
        - Adds a new device to the `deviceList`.
    - `displayDevices()`:
        - Loops through the `deviceList` and prints details of each device, its power consumption, and maintenance cost.
    - `calculateTotalMaintenanceCost()`:
        - Loops through `deviceList` and computes the total maintenance cost for all devices.

Tasks:

1. Implement the `Device` class and its concrete subclasses (`Laptop`, `Smartphone`, and `Desktop`) as described.
2. Create several devices:
    - A laptop with a processor power of `45 W`, daily usage of `5 hours`, and a yearly maintenance cost of `$150`.
    - A smartphone with a battery capacity of `4000 mAh`, `300 charge cycles per year`, and a maintenance cost of `$0.5` per cycle.
    - A desktop with a power consumption of `250 W` per hour, daily usage of `8 hours`, and a yearly service cost of `$200`.
3. Add all devices to the `DeviceManager` and display the following:
    - Each device's details, power consumption, and maintenance cost.
    - Total maintenance cost for all devices.

QUESTION 4

You are tasked with creating a **2D Shape Management System** for a design application that allows users to manage, resize, and render various 2D shapes. Use **abstraction** to define the core operations for all shapes and provide specific implementations for different shape types.

Abstract Class: Shape2D

- Fields:
    - `color` (String): The color of the shape.
    - `positionX` (double): The X-coordinate of the shape's position.
    - `positionY` (double): The Y-coordinate of the shape's position.
- Constructor:
    - A three-argument constructor to initialize `color`, `positionX`, and `positionY`.
- Abstract Methods:
    - `draw()`:
        - Outputs a representation of the shape being drawn with its color and position.
    - `resize(double factor)`:
        - Resizes the shape by a given factor.
- Concrete Method:
    - `move(double deltaX, double deltaY)`:
        - Updates the `positionX` and `positionY` fields based on the delta values

Concrete Class: Rectangle (Derived from Shape2D)

- Fields:
  - `width` (double): The width of the rectangle.
  - `height` (double): The height of the rectangle.
- Constructor:
  - A five-argument constructor to initialize the `color`, `positionX`, `positionY`, `width`, and `height`.
- Implementations:
  - `draw()`:
    - Outputs: `"Drawing a Rectangle: Color = [color], Position = ([positionX], [positionY]), Width = [width], Height = [height]"`
  - `resize(double factor)`:
    - Updates `width` and `height` by multiplying them by `factor`.

Concrete Class: Circle (Derived from Shape2D)

- Fields:
  - `radius` (double): The radius of the circle.
- Constructor:
  - A four-argument constructor to initialize the `color`, `positionX`, `positionY`, and `radius`.
- Implementations:
  - `draw()`:
    - Outputs: `"Drawing a Circle: Color = [color], Position = ([positionX], [positionY]), Radius = [radius]"`
  - `resize(double factor)`:
    - Updates `radius` by multiplying it by `factor`

Class: ShapeManager

- Fields:
  - `shapes` (List of Shape2D): Stores all shapes.
- Methods:
  - `addShape(Shape2D shape)`:

        - Adds a new shape to the `shapes` list.
    - `drawAllShapes()`:
        - Loops through all shapes and calls their `draw()` method.
    - `resizeAllShapes(double factor)`:
        - Loops through all shapes and calls their `resize()` method with the given factor.
    - `moveShape(String shapeType, double deltaX, double deltaY)`:
        - Finds all shapes of the specified type and moves them by the delta values using the `move()` method.

Tasks:

1. Implement the `Shape2D` class and its subclasses (`Rectangle` and `Circle`) as described.
2. Create multiple shapes:
    - A rectangle with color `red`, position `(2, 3)`, width `5.0`, and height `10.0`.
    - A circle with color `blue`, position `(4, 5)`, and radius `7.0`.
    - Another rectangle with color `green`, position `(6, 8)`, width `3.0`, and height `6.0`.
3. Add all shapes to the `ShapeManager` and perform the following operations:
    - Draw all shapes.
    - Resize all shapes by a factor of `2.0`.
    - Move all rectangles by `(-1.0, 1.0)` and draw them again.

QUESTION 5

You are tasked to create a **University Management System** focusing on the **Hostel and Department Management**. Extend the concept to accommodate multiple departments, hostels, and students, and introduce a menu-driven driver class for handling advanced operations.

Interface: Department

- **Attributes**:
    - `deptName` (String): Name of the department.
    - `deptHead` (String): Name of the department head.
- **Methods**:
    - `printDepartmentDetails()`: Prints the department details.

Class: Hostel

- **Attributes**:
  - `hostelName` (String): Name of the hostel.
  - `hostelLocation` (String): Location of the hostel.
  - `numberOfRooms` (int): Number of rooms in the hostel.
- **Methods**:
  - `setHostelDetails(String name, String location, int rooms)`: Sets the hostel details.
  - `printHostelDetails()`: Prints the hostel details.

Class: Student (Extends `Hostel`, Implements `Department`)

- **Attributes**:
  - `studentName` (String): Name of the student.
  - `regdNo` (String): Registration number of the student.
  - `electiveSubject` (String): Elective subject chosen by the student.
  - `avgMarks` (double): Average marks of the student.
- **Methods**:
  - `getStudentDetails()`:
    - Input all student-related details, including department and hostel details.
  - `printStudentDetails()`:
    - Print all student details, including department and hostel details.
  - Implement abstract methods from `Department`:
    - `printDepartmentDetails()`

Class: UniversityDriverMenu

- **Attributes**:
  1. `studentList` (ArrayList<Student>): A dynamic list to store students.
- **Menu Options**:
  1. **Admit New Student**:
     - Add a new student by entering all required details.
  2. **Migrate a Student**:
     - Modify the hostel details of an existing student based on their registration number.
  3. **Display Student Details**:
     - Search for a student by `regdNo` and display their details.