OOP Principles Scenario Problems

---

1) Bank Account Management

Learning goals: encapsulation, inheritance, overriding, subclass polymorphism.

Classes & hierarchy

- BankAccount (base): private String accountNumber, private double balance.
  - getBalance(), setBalance(double) (validate non-negative), deposit(double), withdraw(double), String summary().
  - double calculateInterest(int months) → **to be overridden**; default returns 0.

- SavingsAccount extends BankAccount:
  - private double annualRate (e.g., 4%).
  - Override calculateInterest(int months) using **monthly compounding**:
    - interest = balance * (Math.pow(1 + annualRate/12, months) - 1).

- CurrentAccount extends BankAccount:
  - private double monthlyFee (e.g., 100.0), private double annualRate (e.g., 1%).
  - Override calculateInterest(int months) using **simple interest**:
    - interest = balance * (annualRate/12) * months - monthlyFee * months (can be negative if fees exceed interest).

Main tasks

1. Create a SavingsAccount (rate 0.04) and a CurrentAccount (rate 0.01, fee 100). Seed with opening balances via constructor (use super for accountNumber/balance).
2. Deposit and withdraw valid/invalid amounts (show validation via setters/methods: no negative deposit; no overdraft unless balance allows).
3. For both accounts, compute and print 6-month interest using overridden methods; print **projected balance = balance + interest**.
4. Store both in a BankAccount[] and iterate to print polymorphic summary() and calculateInterest(6) results.

2) Employee Hierarchy

Learning goals: encapsulation, inheritance, super constructor, overriding, subclass polymorphism.

Classes & hierarchy

- **Employee (base class):**
  - **Private fields:** String name, double baseSalary**.**
  - Provide **constructor** Employee(String name, double baseSalary)**.**
  - Provide **getters and setters** (validate: baseSalary >= 0).
  - **Method String role()** → returns "Employee"**.**
  - **Method double totalComp()** → returns baseSalary**.**

- **Manager extends Employee:**
  - **Additional private fields:** String department, double allowance.
  - **Constructor Manager(String name, double baseSalary, String department, double allowance)** → call super(name, baseSalary).
  - **Override role()** → return "Manager".
  - **Override totalComp()** → return baseSalary + allowance.

- **Developer extends Employee:**
  - **Additional private fields:** String language, double bonus.
  - **Constructor Developer(String name, double baseSalary, String language, double bonus)** → call super(name, baseSalary).
  - **Override role()** → return "Developer".
  - **Override totalComp()** → return baseSalary + bonus.

Main tasks

1. Create one Manager and two Developer objects (details can be hardcoded or taken from Scanner).
   - Use the super constructor to initialize name and base salary.
2. Store them in an Employee[] array.
3. Iterate through the array and print each employee's role() and totalComp() polymorphically (i.e., using base class reference).
4. Use the setter to update one employee's base salary (e.g., give the developer a raise).
   - Print salary before and after the update to demonstrate encapsulation.
5. Show how overridden methods are invoked depending on the actual object type, even when referenced by Employee[].

3) Vehicle Rental System

Learning goals: overriding, subclass polymorphism, basic validation.

Classes & hierarchy

- **Vehicle** (base class):
  - Private fields: String plate, double baseRate.
  - Constructor: initialize plate and baseRate.
  - Getters/setters with validation (e.g., baseRate >= 0).
  - Method double quote(int days) → default implementation: days * baseRate. If days <= 0, return 0.
  - Method String type() → return "Vehicle".

- **Car** extends Vehicle:
  - Override quote(int days): if days >= 7, apply **10% discount** → 0.9 * days * baseRate. Otherwise, use normal calculation.
  - Override type() → return "Car".

- **Bike** extends Vehicle:
  - Override quote(int days): cost = 0.85 * days * baseRate (always 15% cheaper).
  - Override type() → return "Bike".

- **Truck** extends Vehicle:
  - Override quote(int days): cost = days * baseRate + 500 (flat heavy fee added).
  - Override type() → return "Truck".

Main tasks

1. Create one object each of Car, Bike, and Truck with appropriate plate numbers and base rates (hardcoded or user input).
2. Store them in a Vehicle[] array.
3. For days = 3 and days = 7:
   - Iterate through the array.
   - Print type() and quote(days) polymorphically.
   - Show how different rules apply to each vehicle type.
4. Demonstrate **validation**: attempt days = 0 (or negative). The method should handle it gracefully by returning 0 as the rental cost.
5. Highlight polymorphism: although the array is of type Vehicle[], the correct overridden quote() is called for each subclass.

4) Library Book Management

Learning goals: encapsulation, constructor overloading, basic state transitions.

Classes & hierarchy

- **Book**
  - Private fields:
    - String title
    - String author
    - boolean issued
  - Constructors:
    - Book(String title) → initializes book with only title; author can be "Unknown".
    - Book(String title, String author) → initializes book with title and author.
    - By default, issued = false.
  - Getters and setters for all fields.
  - Method void issue() → changes state:
    - If issued == false, set issued = true.
    - If already issued, print a friendly error: "Book already issued: <title>".
  - Method void returnBook() → changes state:
    - If issued == true, set issued = false.
    - If not issued, print a friendly error: "Book is not issued: <title>".
  - Method String getDetails() → return formatted info: "Title: ..., Author: ..., Issued: true/false".

Main tasks

1. Create 3 Book objects using different constructors:
   - One with only title.
   - One with title and author.
   - Another with only title.
2. Print their initial details (all should be issued = false).
3. Issue two books using issue().
4. Attempt to issue a book that is already issued (should print the error message).
5. Return one of the issued books using returnBook().
6. Print the details of all books again, showing which are currently issued and which are available.

5) Shape Area Calculator

Learning goals: method overriding, method overloading, polymorphism, arrays.

Classes & hierarchy

- **Shape** (base class)
  - Methods:
    - double area() → default implementation returns 0.
    - String name() → returns "Shape".
- **Rectangle** extends Shape
  - Fields: int length, int width.
  - Constructors:
    - Rectangle(int length, int width) → initializes rectangle.
    - Rectangle() → default constructor (optional).
  - Overloaded area() methods:
    - int area(int side) → calculates square area (side*side).
    - int area(int l, int w) → calculates rectangle area (l*w).
  - Override double area() → calculate area using the rectangle's length and width.
  - Override String name() → returns "Rectangle".
- **Circle** extends Shape
  - Field: double radius.
  - Constructor: Circle(double radius).
  - Override double area() → Math.PI * radius * radius.
  - Override String name() → returns "Circle".
- **Triangle** extends Shape
  - Fields: double base, double height.
  - Constructor: Triangle(double base, double height).
  - Override double area() → 0.5 * base * height.
  - Override String name() → returns "Triangle".

Main tasks

1. Create a Shape[] array containing:
   - new Rectangle(4, 6)
   - new Circle(3)
   - new Triangle(5, 2)
2. Iterate through the array and print each shape's name() and area() **polymorphically**.

3. Demonstrate **overloading** for the Rectangle class:
    - Call new Rectangle().area(5) → square.
    - Call new Rectangle().area(3, 7) → rectangle.
4. Observe the difference between **overridden methods** (polymorphic behavior via Shape[]) and **overloaded methods** (compile-time method selection).

6) School System

Learning goals: encapsulation in base class, inheritance, method overriding, polymorphism.

Classes & hierarchy

- **Person** (base class)
    - Private fields: String name, int age.
    - Constructor: Person(String name, int age).
    - Getters and setters (validate 0 < age < 120).
    - Method String describe() → returns a basic string with name and age.

- **Student** extends Person
    - Field: String grade.
    - Constructor: Student(String name, int age, String grade) → call super(name, age).
    - Override describe() → include grade in the returned string.

- **Teacher** extends Person
    - Field: String subject.
    - Constructor: Teacher(String name, int age, String subject) → call super(name, age).
    - Override describe() → include subject in the returned string.

Main tasks

1. Create one Student and one Teacher using constructors with super.
2. Use setter to change age (test validation for invalid and valid ages).
3. Store both objects in a Person[] array.
4. Iterate through the array and print describe() for each object **polymorphically** to show overridden behavior.

7) Online Payment System

Learning goals: runtime polymorphism, method overriding.

Classes & hierarchy

- **Payment** (base class)
  - Method: String pay(double amount) → returns a generic receipt text with the amount.
- **CreditCardPayment** extends Payment
  - Override pay(double amount) → apply 2% fee (amount * 1.02) and return receipt with total.
- **PayPalPayment** extends Payment
  - Override pay(double amount) → apply 3% fee plus 5 flat (amount * 1.03 + 5) and return receipt with total.
- **CashPayment** extends Payment
  - Override pay(double amount) → exact amount, return receipt.

Main tasks

1. Create one object of each payment type (CreditCardPayment, PayPalPayment, CashPayment).
2. Store them in a Payment[] array.
3. Call pay(1000) on each element and print the returned receipts.
4. Compare totals to demonstrate **polymorphic behavior** with overridden methods.

8) Animal Sound Simulation

Learning goals: method overriding, polymorphic dispatch, static helper methods.

Classes & hierarchy

- **Animal** (base class)
  - Method: String makeSound() → returns a generic sound like "Some sound".
- **Dog** extends Animal
  - Override makeSound() → return "Woof".

- **Cat** extends Animal
  - Override makeSound() → return "Meow".
- **Cow** extends Animal
  - Override makeSound() → return "Moo".

Main tasks

1. Create a mixed array Animal[] containing multiple Dog, Cat, and Cow objects.

2. Loop through the array and print the sound of each animal using makeSound() to demonstrate **polymorphic dispatch**.

3. Add a static helper method static void chorus(Animal[] animals) in the main class.

4. Call chorus(animals) to make all animals in the array “sing” together, printing each sound to show runtime polymorphism.

9) University Staff System

Learning goals: encapsulation, inheritance, method overriding, subclass specialization, polymorphism.

Classes & hierarchy

- **Staff** (base class)
  - Private fields: int id, String name.
  - Constructor: Staff(int id, String name).
  - Getters and setters for both fields (validate id > 0 and name not empty).
  - Method String displayRole() → returns "Staff" with name and id.
- **Professor** extends Staff
  - Field: String field (area of specialization).
  - Constructor: Professor(int id, String name, String field) → call super(id, name).
  - Override displayRole() → include field in returned string, e.g., "Professor <name>, ID: <id>, Field: <field>".
- **Clerk** extends Staff
  - Field: String desk (desk number/assignment).
  - Constructor: Clerk(int id, String name, String desk) → call super(id, name).
  - Override displayRole() → include desk info, e.g., "Clerk <name>, ID: <id>, Desk: <desk>".

Main tasks

1. Create **one Professor** and **one Clerk** using constructors with super.
2. Use the setter to **change the name** of one staff member; print displayRole() for both before and after to demonstrate **encapsulation and validation**.
3. Store both objects in a Staff[] array and iterate to print displayRole() **polymorphically**, showing overridden behavior.
4. Add a method static void staffSummary(Staff[] staffMembers) in the main class to print all staff roles in a formatted list.
5. Create a **second Professor and Clerk** with different IDs and fields/desks. Update the Staff[] array and use the staffSummary() method again to display all staff.
6. Bonus challenge: demonstrate **typecasting** by checking if an element in the Staff[] array is a Professor or Clerk, then print extra details specific to that subclass.

10) Calculator

Learning goals: method overloading, API design, type resolution, compile-time polymorphism.

Class & methods

- **Calculator**
  - Overloaded methods:
    - int add(int a, int b) → sum of two integers.
    - double add(double a, double b) → sum of two doubles.
    - int add(int a, int b, int c) → sum of three integers.
    - Optional: int add(int[] nums) → sum of all integers in the array.
  - Ensure proper **method signatures** so compiler resolves the correct overload.

Main tasks

1. Create a Calculator object.
2. Call all overloaded methods with appropriate arguments:
   - add(2, 3)
   - add(2.5, 3.7)
   - add(1, 2, 3)
   - add(new int[]{1,2,3,4,5}) (optional).
3. Print results of each call.
4. Demonstrate **type resolution** by calling with mixed literals, e.g., add(5, 6.0) → show how casting or implicit conversion chooses the correct overload.
5. Bonus: Try calling add(5, 6, 7.0) and explain why it causes a **compile-time error**, reinforcing understanding of overload resolution rules.
6. Optional enhancement: Add a method double add(double[] nums) to handle an array of doubles and demonstrate overloading with array inputs.

11) E-commerce Product System

Learning goals: encapsulation, inheritance, method overriding, polymorphism, array handling.

Classes & hierarchy

- **Product** (base class)
  - Private fields: int id, String name, double price.
  - Constructor: Product(int id, String name, double price).
  - Getters and setters (validate: price >= 0).
  - Method String details() → returns formatted string with id, name, and price.
- **Electronics** extends Product
  - Field: int warrantyMonths.
  - Constructor: Electronics(int id, String name, double price, int warrantyMonths) → call super(id, name, price).
  - Override details() → append warranty info, e.g., "Warranty: 24 months".
- **Clothing** extends Product
  - Field: String size.
  - Constructor: Clothing(int id, String name, double price, String size) → call super(id, name, price).
  - Override details() → append size info, e.g., "Size: M".

Main tasks

1. Create one Electronics and one Clothing object using constructors with super.
2. Store both objects in a Product[] array and iterate to print details() **polymorphically**.
3. Apply a **discount** by changing the price via setter (e.g., 10% off) and reprint details().
4. Demonstrate **encapsulation** by attempting invalid price updates (negative values) and showing that validation prevents it.
5. Bonus tasks:
   - Add a **method applyDiscount(double percent)** in Product class and override in subclasses if needed.
   - Create additional Electronics and Clothing objects, store in the array, and print all details sorted by price (optional: demonstrates array iteration and basic logic).

12) Transport Ticket Booking

Learning goals: method overriding, runtime polymorphism, polymorphic arrays, simple fare calculation.

Classes & hierarchy

- **Transport** (base class)
  - Method: String bookTicket(String from, String to) → returns a generic receipt including from/to cities.
- **Bus** extends Transport
  - Fare calculation: base fare per km = 1.0.
  - Override bookTicket(String from, String to) → compute distance using a small helper map, calculate total fare, generate seat code, and return formatted receipt.
- **Train** extends Transport
  - Fare calculation: base fare per km = 0.5.
  - Override bookTicket(String from, String to) → similar logic as Bus, with different fare rules and seat code format.
- **Flight** extends Transport
  - Fare calculation: base fare per km = 3.0 + airport tax 500.
  - Override bookTicket(String from, String to) → calculate total fare including tax, generate seat code, and return receipt.
- **Helper**: a static method int computeDistance(String from, String to) returning a fixed distance map (e.g., CityA-CityB = 100 km).

Main tasks

1. Create one object each of Bus, Train, and Flight.
2. Call bookTicket("CityA", "CityB") on each object and print the receipts.
3. Store all objects in a Transport[] array and iterate to call bookTicket polymorphically.
4. Demonstrate **overriding**: verify that each subclass generates its own fare, seat code, and receipt format.
5. Add **bonus tasks**:
   - Try booking tickets for multiple city pairs using a loop.
   - Extend the seat code generation to ensure unique codes per booking.
   - Print a summary of total fares collected for all transports (introduces array aggregation logic).
6. Optional: handle invalid city input gracefully (e.g., city not in map) and return an error message without crashing.

13) Sports Team Management

Learning goals: encapsulation, inheritance, method overriding, polymorphism, arrays.

Classes & hierarchy

- **Player** (base class)
  - Private fields: String name, int age.
  - Constructor: Player(String name, int age).
  - Getters and setters (validate: age > 0).
  - Method String stats() → returns a basic string with player name and age.
- **CricketPlayer** extends Player
  - Fields: int runs, int wickets.
  - Constructor: CricketPlayer(String name, int age, int runs, int wickets) → call super(name, age).
  - Override stats() → include runs and wickets in the returned string.
- **FootballPlayer** extends Player
  - Fields: int goals, int assists.
  - Constructor: FootballPlayer(String name, int age, int goals, int assists) → call super(name, age).
  - Override stats() → include goals and assists in the returned string.

Main tasks

1. Build a small squad with 2 CricketPlayer and 2 FootballPlayer objects using constructors.
2. Store all players in a Player[] array.
3. Update some stats (e.g., runs, wickets, goals, assists) using setters and print **before/after** to show encapsulation.
4. Iterate through the array and call stats() **polymorphically** for each player, demonstrating overridden behavior.
5. Bonus tasks:
   - Add a method static void teamSummary(Player[] squad) to print all player stats in a formatted table.
   - Calculate totals: sum of runs for cricket players, total goals for football players.
   - Demonstrate **typecasting**: check if a Player is a CricketPlayer or FootballPlayer and print an additional message like "This player is a cricket specialist".
6. Optional: allow adding a **new player dynamically** via Scanner input and update the array (practice with arrays and object references).

14) Smart Device Hierarchy

Learning goals: inheritance, method overriding, method overloading, polymorphism, arrays.

Classes & hierarchy

- **Device** (base class)
  - Methods: void turnOn() → prints "Device powering on..."; void turnOff() → prints "Device shutting down...".
- **Phone** extends Device
  - Overloaded methods:
    - void call(String number) → prints "Calling <number>...".
    - void call(String number, int minutes) → prints "Calling <number> for <minutes> minutes...".
  - Override turnOn() → display "Phone booting with animation...".
- **Laptop** extends Device
  - Override turnOn() → display "Laptop starting OS...".

Main tasks

1. Create a Device[] array containing one Phone and one Laptop.
2. Iterate through the array and call turnOn() **polymorphically**, showing overridden behavior.
3. Demonstrate **overloading** by calling both versions of call() on the Phone object.
4. Call turnOff() on all devices to show consistent base behavior.
5. Bonus tasks:
   - Add a static helper method static void deviceStatus(Device[] devices) to print the type and current state of each device.
   - Try adding another subclass Tablet with its own override of turnOn() and a new method browse(String url).
   - Demonstrate typecasting: check if a Device is a Phone in the array and then call the overloaded call() method.
6. Optional: simulate a small “smart device startup sequence” by iterating through multiple devices and calling turnOn() in a loop, showing polymorphism in action.

15) Employee Bonus Calculation

Learning goals: method overriding, runtime polymorphism, subclass-specific logic, arrays, conditional logic.

Classes & hierarchy

- **Employee** (base class)
  - Private field: String name.
  - Protected field: double baseSalary.
  - Constructor: Employee(String name, double baseSalary).
  - Getters for both fields.
  - Method double bonus() → default returns 0.
  - Method String displayCompensation() → returns a formatted string with name, baseSalary, and bonus().
- **PermanentEmployee** extends Employee
  - Override bonus() → return baseSalary * 0.20.
- **ContractEmployee** extends Employee
  - Override bonus() → if baseSalary > 30000, return 5000; else return 2000.

Main tasks

1. Create one PermanentEmployee and one ContractEmployee with initial base salaries.
2. Store both objects in an Employee[] array.
3. Iterate through the array and print each employee's displayCompensation() to show base salary and computed bonus **polymorphically**.
4. Adjust salaries using setters (e.g., increase one above/below the threshold for ContractEmployee) and re-compute bonuses to demonstrate conditional logic.
5. Bonus tasks:
   - Add a static method static void companyBonusReport(Employee[] staff) to compute total bonuses for all employees.
   - Demonstrate **typecasting**: identify ContractEmployee and PermanentEmployee in the array to print a subclass-specific message.
   - Add an optional method increaseSalary(double percent) in Employee and override in subclasses if needed, then show updated compensation after increases.
6. Optional: Create multiple employees of each type, store in an array, and sort them by total compensation before printing the report.

16) Banking Loan System

Learning goals: encapsulation, inheritance, method overriding, polymorphism, input validation.

Classes & hierarchy

- **Loan** (base class)
  - Private fields: double amount, double rate.
  - Constructor: Loan(double amount, double rate).
  - Getters and setters (validate amount > 0 and rate >= 0).
  - Method double calculateEMI() → default returns 0.
  - Method String loanDetails() → return formatted string with amount and rate.
- **HomeLoan** extends Loan
  - Override calculateEMI() → use formula: EMI = amount * rate / 12 (simplified).
  - Optional: add field int tenureYears and include in EMI calculation using EMI = (amount * rate / 12) / (1 - Math.pow(1 + rate / 12, -tenureYears*12)).
- **CarLoan** extends Loan
  - Override calculateEMI() → slightly different formula or add fixed processing fee, e.g., EMI = amount * rate / 12 + 100.

Main tasks

1. Input loan details (amount and rate) for one HomeLoan and one CarLoan.
2. Create objects using constructors and store in a Loan[] array.
3. Iterate through the array and print each loan's loanDetails() and computed calculateEMI() **polymorphically**.
4. Demonstrate **encapsulation** by trying to set invalid values for amount/rate and show that validation prevents incorrect data.
5. Bonus tasks:
   - Add a static method static void displayLoanSummary(Loan[] loans) to print all loan types and EMI in a formatted table.
   - Add optional field tenureMonths to both subclasses and calculate EMI based on tenure.
   - Demonstrate **typecasting**: identify HomeLoan vs CarLoan in the array and print a subclass-specific message (e.g., "Home loan with benefits").
6. Optional: allow input of multiple loans of each type and compute total EMI for the portfolio.

17) Online Store Checkout

Classes & hierarchy

- **Checkout** (base class)
  - Method: double calculateBill() → returns total bill (default 0 or base implementation).
  - Method: String checkoutDetails() → return a generic receipt string.
- **OnlineCheckout** extends Checkout
  - Override calculateBill() → compute total for online orders, e.g., sum of item prices + standard shipping fee.
  - Overloaded methods for different payment types:
    - double calculateBill(String creditCardNumber) → apply online discount or surcharge.
    - double calculateBill(String debitCardNumber, boolean applyReward) → apply optional reward points or cashback.
  - Override checkoutDetails() → include payment method and final amount.
- **OfflineCheckout** extends Checkout
  - Override calculateBill() → compute total for in-store purchase; may include tax but no shipping.
  - Override checkoutDetails() → include store location, payment type, and final amount.

Main tasks

1. Create one OnlineCheckout and one OfflineCheckout object.
2. Store them in a Checkout[] array.
3. Iterate through the array and call calculateBill() polymorphically; print checkoutDetails() for each.
4. Demonstrate **overloading** on OnlineCheckout by calling both overloaded calculateBill() methods (creditCard and debitCard versions) and printing results.
5. Bonus tasks:
   - Allow input of multiple items with prices via Scanner, compute total dynamically.
   - Add optional methods to apply discounts, rewards, or shipping fees in the OnlineCheckout subclass.
   - Demonstrate **typecasting**: identify which element in Checkout[] is OnlineCheckout or OfflineCheckout to apply subclass-specific logic.
6. Optional: Extend with GiftCheckout subclass that overrides calculateBill() to include gift wrapping fee and show polymorphic behavior in the array.

18) University Course Management

Learning goals: inheritance, method overriding, runtime polymorphism, arrays, subclass specialization.

Classes & hierarchy

- **Course** (base class)
  - Fields: String courseName, int courseCode.
  - Constructor: Course(String courseName, int courseCode).
  - Method String getDetails() → return basic course information (name and code).
- **OnlineCourse** extends Course
  - Fields: String platform, int durationWeeks.
  - Constructor: OnlineCourse(String courseName, int courseCode, String platform, int durationWeeks) → call super(courseName, courseCode).
  - Override getDetails() → include platform and duration information.
- **OfflineCourse** extends Course
  - Fields: String classroom, String schedule.
  - Constructor: OfflineCourse(String courseName, int courseCode, String classroom, String schedule) → call super(courseName, courseCode).
  - Override getDetails() → include classroom and schedule details.

Main tasks

1. Create multiple objects of OnlineCourse and OfflineCourse using constructors.
2. Store them in a Course[] array of base class references.
3. Iterate through the array and print getDetails() **polymorphically** for each course.
4. Bonus tasks:
   - Add a static method static void courseSummary(Course[] courses) to print all course details in a formatted table.
   - Demonstrate **typecasting**: identify whether a Course is an OnlineCourse or OfflineCourse and print subclass-specific messages, e.g., "Platform: Coursera" or "Classroom: B101".
   - Allow updating fields like durationWeeks or schedule via setters and show updated details.
5. Optional enhancement: add a method double calculateFees() in Course, override in subclasses to compute tuition differently for online vs offline courses.

19) Hospital System

Learning goals: encapsulation, inheritance, method overriding, polymorphism, arrays.

Classes & hierarchy

- **Person** (base class)
  - Private fields: String name, int age.
  - Constructor: Person(String name, int age).
  - Getters and setters (validate: age > 0 and name not empty).
  - Method String displayInfo() → returns basic info: "Name: <name>, Age: <age>".
- **Patient** extends Person
  - Field: String disease.
  - Constructor: Patient(String name, int age, String disease) → call super(name, age).
  - Override displayInfo() → include disease information.
- **Doctor** extends Person
  - Field: String specialization.
  - Constructor: Doctor(String name, int age, String specialization) → call super(name, age).
  - Override displayInfo() → include specialization information.

Main tasks

1. Create multiple Patient and Doctor objects using constructors.
2. Store all objects in a Person[] array of base class references.
3. Iterate through the array and print displayInfo() for each object **polymorphically**.
4. Demonstrate **encapsulation** by attempting to update private fields via setters, including invalid values to show validation.
5. Bonus tasks:
   - Add a static helper method static void hospitalSummary(Person[] staffAndPatients) to print all information in a formatted table.
   - Demonstrate **typecasting**: check if a Person is a Patient or Doctor in the array and print additional subclass-specific messages (e.g., "Patient requires special care" or "Doctor available for consultation").
   - Add an optional method in Doctor like consult(Patient p) which prints a message including doctor name, patient name, and disease.
6. Optional enhancement: allow dynamic addition of new patients and doctors via Scanner input and update the Person[] array.

20) Vehicle Workshop

Learning goals: method overriding, method overloading, polymorphism, arrays, subclass specialization.

Classes & hierarchy

- **Workshop** (base class)
  - Method: void repair() → prints "General vehicle repair in progress...".
- **CarWorkshop** extends Workshop
  - Override repair() → print "Car repair started...".
  - Overloaded methods:
    - void repair(String partName) → print "Repairing <partName> in car...".
    - void repair(String partName, int cost) → print "Repairing <partName> in car with cost <cost>".
- **BikeWorkshop** extends Workshop
  - Override repair() → print "Bike repair started...".

Main tasks

1. Create one CarWorkshop and one BikeWorkshop object.
2. Store them in a Workshop[] array of base class references.
3. Iterate through the array and call repair() polymorphically to demonstrate **overriding**.
4. Demonstrate **overloading** by calling both repair(String partName) and repair(String partName, int cost) on the CarWorkshop object.
5. Bonus tasks:
   - Add a static helper method static void performWorkshop(Workshop[] workshops) to iterate and call repair() on all vehicles.
   - Try calling the overloaded repair() method with invalid cost (negative value) to show simple validation.
   - Optional: Add a method estimateCost(String[] parts, int[] costs) in CarWorkshop to calculate total repair cost for multiple parts.
6. Optional enhancement: extend BikeWorkshop with its own overloaded method repair(String partName, boolean urgent) to handle urgent repairs and show polymorphic behavior.