**Q1: Product Price Comparison**

**Scenario:**  
You are designing an **e-commerce system** that requires comparing two products based on **price** and **name**. Implement a Product class with attributes name, price, and an equals method to compare two products.

* Override equals() to compare products by **both name and price**.
* Ensure correct object instantiation and comparison.

**Example:**

Product p1 = new Product("Laptop", 50000);

Product p2 = new Product("Laptop", 50000);

System.out.println(p1.equals(p2)); // true

**2. Inheritance (Single, Multilevel, Hierarchical)**

**Q2: Vehicle Inheritance System**

**Scenario:**  
You are designing a **vehicle management system** with multiple vehicle types:

1. Vehicle (parent class) → contains common attributes (speed, fuelType).
2. Car (inherits from Vehicle) → has an additional numDoors attribute.
3. ElectricCar (inherits from Car) → adds batteryCapacity attribute.

* Implement **single and multilevel inheritance**.
* Define a method displayDetails() in Vehicle and override it in Car and ElectricCar.

**Example Usage:**

ElectricCar tesla = new ElectricCar(120, "Electric", 4, 80);

tesla.displayDetails();

**Expected Output:**

Speed: 120 km/h, Fuel Type: Electric, Doors: 4, Battery: 80 kWh

**3. Polymorphism (Method Overloading, Method Overriding, Constructor Chaining)**

**Q3: Payment System using Method Overloading**

**Scenario:**  
You are building a **payment processing system** that allows users to **pay in different ways**. Implement a Payment class with the following overloaded methods:

* pay(double amount): Pay using a default payment method.
* pay(double amount, String method): Pay using a specific method (e.g., "Credit Card", "UPI").
* pay(double amount, String method, int emiMonths): Pay using **EMI** with a duration.
* Implement **method overloading** for pay().
* Demonstrate different ways to pay.

**Example Usage:**

Payment p = new Payment();

p.pay(1000); // Default payment

p.pay(2000, "Credit Card");

p.pay(5000, "UPI", 12);

**Q4: Employee Salary Calculation using Method Overriding**

**Scenario:**  
You are designing a **salary system** with:

1. Employee (base class) → method calculateSalary() that returns base salary.
2. Manager (inherits from Employee) → overrides calculateSalary() to **add a bonus**.

* Implement **method overriding** in Manager class.

**Example Usage:**

Employee emp = new Employee();

Manager mgr = new Manager();

System.out.println(emp.calculateSalary()); // Base salary: 50000

System.out.println(mgr.calculateSalary()); // Base salary: 50000 + Bonus: 10000

**Q5: Constructor Chaining in a Banking System**

**Scenario:**  
You are designing a BankAccount class that should allow:

1. **Default account creation** (balance = 0).
2. **Account creation with balance**.
3. **Account creation with balance and owner name**.

* Implement **constructor chaining** to avoid code repetition.

**Example Usage:**

BankAccount acc1 = new BankAccount();

BankAccount acc2 = new BankAccount(1000);

BankAccount acc3 = new BankAccount(5000, "John Doe");

**4. Encapsulation (Access Modifiers, Getter/Setter Methods)**

**Q6: Secure User Data with Encapsulation**

**Scenario:**  
Your company needs a **secure user management system** where:

* The User class has private fields (username, password).
* Only **getter** for username, but **no getter for password**.
* A **setter for password** that ensures it meets security criteria.
* Implement proper **access control** using getters/setters.
* Ensure password must be **at least 8 characters long**.

**5. Creating and Using Packages, Static Imports**

**Q7: Organizing a Library Management System**

**Scenario:**  
You are building a **library management system** with separate modules:

1. library.books.Book → Class for storing book details.
2. library.members.Member → Class for storing member details.
3. library.utils.Utility → Class with a static method generateId().

* Implement **packages** for modular design.
* Use **static import** to call generateId() without the class name.

**6. Abstraction (Abstract Classes, Interfaces)**

**Q8: Online Shopping System using Abstraction**

**Scenario:**  
You are designing an **online shopping system** where:

1. Product is an **abstract class** with an abstract method calculateDiscount().
2. Electronics and Clothing extend Product and provide **custom discount logic**.

* Implement an **abstract class** with at least one abstract method.
* Ensure **different discount calculations** for Electronics and Clothing.

**Example Usage:**

Product phone = new Electronics(50000);

Product shirt = new Clothing(2000);

System.out.println(phone.calculateDiscount()); // 10% discount for electronics

System.out.println(shirt.calculateDiscount()); // 5% discount for clothing

**7. Inner Classes (Member, Local, Anonymous)**

**Q9: Using Inner Classes for Authentication**

**Scenario:**  
You are implementing an **authentication system** where:

1. Authenticator class contains a **private inner class** Validator.
2. Validator checks if a given **password is correct**.
3. Only Authenticator can create an instance of Validator.

* Use a **member inner class** to encapsulate validation logic.

**Example Usage:**

Authenticator auth = new Authenticator();

auth.login("admin123"); // Output: Login Successful / Incorrect Password

**Q10: Implementing an Anonymous Class for Sorting**

**Scenario:**  
You need to **sort a list of students** based on marks, but without creating a separate comparator class.

* Use an **anonymous inner class** to implement Comparator<Student>.

**Example Usage:**

List<Student> students = Arrays.asList(

new Student("Alice", 85),

new Student("Bob", 90),

new Student("Charlie", 78)

);

Collections.sort(students, /\* Use Anonymous Class for sorting \*/);