Java Lab Test
Total: 40 Marks

January 2026

Instructions

- Write all classes in **Java**.
- Use proper naming conventions, and code formatting.
- Demonstrate all required concepts clearly in the `main()` method.
- You must submit a **PDF** containing your code, input and output.

1 Task 1

You are developing a simple library management system.
Create a class `Book` with:

- **Private fields:** `title` (String), `author` (String), `yearPublished` (int)
- A **default constructor** that sets "Unknown Title", "Unknown Author", 0
- A **parameterized constructor** that accepts all three values
- A method `displayInfo()` that prints the book details in this format:
  `Title: [title] | Author: [author] | Published: [year]`
- Use encapsulation (private fields + public getters)

In the `main` method, create two `Book` objects (one using default constructor, one using parameterized) and display their information.

2 Task 2

A company wants to track how many employees are currently working.

Create a class `Employee` with:

- Instance fields: `name` (String), `id` (int)
- Static field: `totalEmployees` (int) initialized to 0
- A parameterized constructor that takes name and id, increments `totalEmployees`, and uses `this` keyword
- A method `showDetails()` that prints: `"ID: [id] | Name: [name]"`
- A static method `getTotalEmployees()` that returns the total count

In `main()`, create 3 employees and print each employee's details + total number of employees at the end.

3 Task 3

You are modeling different types of vehicles for a transportation app.

Create a base class `Vehicle` with:

- Fields: `brand` (String), `speed` (int)
- Constructor to initialize them
- Method `move()` that prints `"[brand] is moving at [speed] km/h"`

Create a subclass `Car` that extends `Vehicle` and adds:

- Extra field: `fuelType` (String)
- Override `move()` to print: `"[brand] car is driving at [speed] km/h using [fuelType]"`
- Additional method `honk()` that prints "Beep Beep!"

In `main()`, create one `Vehicle` and one `Car`, then demonstrate method overriding by calling `move()` on both objects (use superclass reference for `Car`).

4 Task 4

Design a hierarchy for electronic devices.

Create classes with multilevel inheritance: `Device` → `Electronics` → `SmartPhone`

- `Device`: field `powerConsumption` (double), method `turnOn()` → "Device is turning on"
- `Electronics`: field `warrantyYears` (int), constructor using `super()`, method `showWarranty()`
- `SmartPhone`: field `osVersion` (String), override `turnOn()` using `super.turnOn()` + add "SmartPhone is booting [osVersion]"

In `main()`, create a `SmartPhone` object and call `turnOn()` and `showWarranty()`.

5 Task 5

A drawing application needs to handle different shapes.
Create a base class `Shape` with:

- **Abstract method** `draw()` (make `Shape` abstract)
- Concrete method `getType()` that returns "Generic Shape"

Create three subclasses (hierarchical):

- `Circle` → override `draw()` → "Drawing a circle"
- `Rectangle` → override `draw()` → "Drawing a rectangle"
- `Triangle` → override `draw()` → "Drawing a triangle"

In `main()`:

- Create an array of `Shape` references (size 4)
- Put different shape objects into the array (polymorphism)
- Loop through the array and call `draw()` on each (demonstrate runtime polymorphism)

6 Task 6

A physics simulation must use constant values that should never change.
Create a **final class** `PhysicsConstants` with:

- `public static final double GRAVITY = 9.81;`
- `public static final double SPEED_OF_LIGHT = 299792458;`
- A final method `showConstants()` that prints both values

Try to create a subclass of `PhysicsConstants` (it should give compile-time error mention it in comment).
In `main()`, call `showConstants()` using class name.

7 Task 7

Scenario: A zoo management system needs animals that can perform different actions.
Create two interfaces:

- `Movable` with method `move()`
- `SoundMaker` with method `makeSound()`

Create an abstract class `Animal` with:

- Field `name`
- Constructor
- Abstract method `eat()`

Create two concrete classes:

- `Dog` extends `Animal` implements `Movable`, `SoundMaker`
- `Bird` extends `Animal` implements `Movable`, `SoundMaker`

Implement all required methods appropriately.
In `main()`, create objects and call all three methods for both animals (show interface polymorphism).

8 Task 8

Scenario: Build a simplified banking system with polymorphism and abstraction.
Create an abstract class `BankAccount` with:

- Protected fields: `accountNumber` (String), `balance` (double)
- Constructor
- Abstract method `withdraw(double amount)` (subtract from balance)
- Concrete method `deposit(double amount)` (add to balance)
- Concrete method `getBalance()`

Create two subclasses:

- `SavingsAccount` extends `BankAccount`
- `CurrentAccount` extends `BankAccount`

Implement the required behavior for each account type.
In `main()`:

- Create array of `BankAccount` (polymorphism)
- Perform deposit/withdraw on both types
- Show final balances