**Practice Questions**

### 1. Beginner Level: The `Student` Class 🎓

**Task:** Create a class named `Student` that encapsulates student data.

**Data Members (Private):**

- `studentId` (String or int)
- `name` (String)
- `gpa` (double or float)

**Methods (Public):**

- **Constructor:** A constructor that initializes all the data members.
- **Getters:** Public getter methods for all data members (e.g., `getStudentId()`, `getName()`, `getGpa()`).
- **Setter:** A public setter method for `gpa` (`setGpa()`). This method must include validation to ensure the GPA value is between 0.0 and 4.0. If an invalid value is provided, the method should print an error message and not update the GPA.

**Goal:** To practice the basics of creating private fields and providing controlled public access with simple validation logic.

### 2. Intermediate Level: The `Car` Class 🚗

**Task:** Design a class named `Car` that models a real-world car, controlling its state through methods.

**Data Members (Private):**

- `make` (String)
- `model` (String)
- `year` (int)
- `speed` (int)

**Methods (Public):**

- **Constructor:** A constructor that initializes `make`, `model`, and `year`. The `speed` must always be initialized to **0**.
- **`getSpeed()`:** A method to return the current speed.
- **`accelerate()`:** A method that increases the `speed` by 5.
- **`brake()`:** A method that decreases the `speed` by 5.
- **Constraint:** The `speed` must never become negative. Ensure your `brake()` method handles this condition. There should be **no public `setSpeed()` method**; speed can only be changed via `accelerate()` and `brake()`.

**Goal:** To understand how encapsulation controls an object's state by forcing interaction through defined behaviors rather than allowing direct modification of data members.

### 3. Advanced Level: The `BankAccount` Class 🏦

**Task:** Create a robust `BankAccount` class with strict rules for data modification.

**Data Members (Private):**

- `accountNumber` (long or String)
- `accountHolderName` (String)
- `balance` (double)

**Methods (Public):**

- **Constructor:** A constructor to initialize the account with an `accountNumber`, `accountHolderName`, and an initial `balance`.
- **Getters:** Public getter methods for all three fields.
- **Immutability:** There should be **no setter for `accountNumber`**. This property should be immutable after the object is created.
- **`deposit(double amount):`** This method should add to the balance. It must validate that the deposit amount is a positive number.
- **`withdraw(double amount):`** This method should subtract from the balance. It must validate two conditions: the withdrawal amount is positive, and the transaction will not result in a negative balance (i.e., check for insufficient funds).

**Goal:** To implement a class with immutable properties (`accountNumber`) and a critical state (`balance`) that is carefully managed through methods with strict validation rules.