7.2 Encapsulation and UML class diagrams

Name: _____ Class: _____ Date: _____ **Total: 21 marks**

KEY WORDS encapsulation • private • public • accessor • mutator • UML class diagram

Objective

Build the skills to answer exam questions on **encapsulation and class design** – private and public members, accessors and mutators with validation, and UML class diagrams.

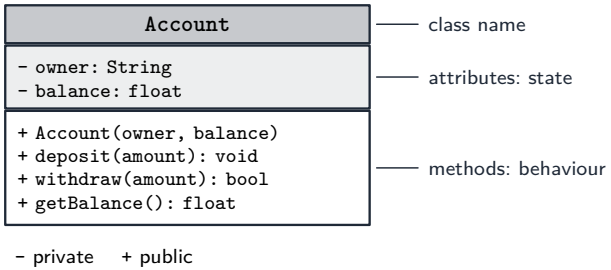
You must be able to:

- explain **encapsulation** and give its advantages
- mark members **private** or **public**, and say who can use each
- write an **accessor** and a **mutator** that rejects invalid values
- draw and read a **UML class diagram** from a scenario

1 Key knowledge and worked examples

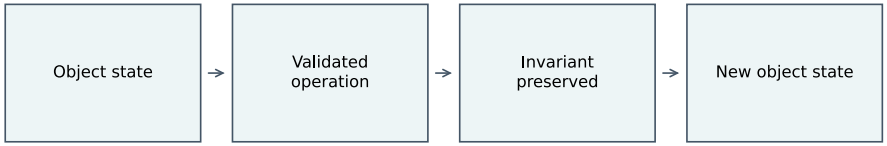
Study these first. Each one shows the method for a question type used later.

■ Key knowledge: a UML class diagram



- Three compartments: name, attributes with types, methods with parameters and return types.
- Attributes are usually private (−); the methods other code needs are public (+).
- In Java: `private double balance;`. In Python a leading underscore marks a private attribute: `self._balance`.

■ Worked example: validation in a mutator



The rule "the balance is never negative" is kept inside the class.

```
class Account:
    def __init__(self, owner, balance):
        self._owner = owner
        self._balance = max(0.0, balance)

    def get_balance(self):              # accessor
        return self._balance

    def withdraw(self, amount):         # mutator
        if amount <= 0 or amount > self._balance:
            return False
        self._balance = self._balance - amount
        return True
```

Call	Returns	_balance after
`acc = Account("Li", 15.0)`		15.0
`acc.withdraw(30.0)`	False	15.0
`acc.withdraw(-5.0)`	False	15.0
`acc.withdraw(15.0)`	True	0.0

1. The check comes **before** the change, so a rejected call changes nothing.
2. No outside code can set `_balance` directly, so every change passes the check.

Worked example: from scenario to class

A gym member has a name and a number of credits. Credits can be added, and a class can be booked if the member has at least one credit, which booking uses up.

1. Nouns that are data: `name`, `credits` → private attributes.
2. Verbs: add credits, book a class → public methods; read credits → an accessor.
3. The rule lives in `book()`: it returns `False` when `credits` is 0.

2 Practice

2.1 Explain what is meant by encapsulation. [2]

2.2 State **two** advantages of making an attribute private and providing a mutator. [2]

2.3 Draw the UML class diagram for the gym member in the worked example, with a constructor, `add_credits(n)`, `book()` and `get_credits()`. [4]

2.4 State the output of `acc = Account("Bo", -20.0)`, then `print(acc.get_balance())`. [1]

3 Exam-style questions

3.1 In a UML class diagram, what does `-` before an attribute mean? [1]

- **A** it is public
- **B** it is private
- **C** it is static
- **D** it can be negative

3.2 Which method is an accessor? [1]

- **A** `set_age(a)`
- **B** `get_age()`
- **C** `birthday()`
- **D** `__init__(name, age)`

3.3 Why should a mutator check a new value **before** changing the attribute? [1]

- **A** checking is faster before the change
- **B** a rejected value then leaves the object unchanged
- **C** accessors cannot read checked values
- **D** the constructor needs the old value

3.4 A school locker system stores a locker's number, whether it is locked and a 4-digit code.

(a) Draw a UML class diagram for `Locker` with a constructor and the public methods `lock()`, `unlock(code)` and `is_locked()`. [4]

(b) Write the method `unlock(code)`. It unlocks the locker and returns `True` only if `code` matches the stored code. [3]

(c) Explain why the code attribute should be private and have **no** accessor. [2]

Solutions

Each answer shows the steps, then the **result**.

2.1

- the data and the methods that act on it are bundled in one class
- the data is hidden and can only be reached through those methods

2.2

- the mutator can reject invalid values, so the object stays valid
- the inside of the class can change without breaking code that uses it

2.3

- name compartment `Member`
- `- name: String`, `- credits: int`
- `+ Member(name)`, `+ add_credits(n): void`, `+ book(): bool`, `+ get_credits(): int`
- `-` on the attributes and `+` on the methods

2.4

- **0.0**: the constructor stores `max(0.0, -20.0)`

3.1 B

- `-` private, `+` public

3.2 B

- it returns a value and changes nothing

3.3 B

- check, then change

3.4

(a)

- name compartment `Locker`
- private attributes with types: `- number: int`, `- locked: bool`, `- code: String`
- constructor `+ Locker(number, code)`
- `+ lock(): void`, `+ unlock(code): bool`, `+ is_locked(): bool`

(b) One answer:

```
def unlock(self, code):
    if code == self._code:
        self._locked = False
        return True
    return False
```

- compares `code` with the stored code
- sets `locked` to `False` only on a match
- returns `True` or `False`

(c)

- other code cannot read the code and open any locker
- the only way in is through `unlock`, which checks the code