Heshawa Cooray

Part A: Conceptual Questions

Definition Encapsulation is the practice of keeping an object's data private and only allowing controlled access through public methods. This prevents unintended modifications and keeps the object's behavior predictable.

Example: A BankAccount class with a private balance variable ensures that the balance cannot be changed directly, only modified through deposit() or withdraw() methods that enforce rules.

Visibility Modifiers

- **Public**: Accessible anywhere.
    - Benefit: Makes code reusable and accessible.
    - Drawback: Can lead to unintended modifications.
- **Private**: Accessible only inside the class.
    - Benefit: Prevents external modification, ensuring data integrity.
    - Drawback: Requires additional methods to access or modify.
- **Protected**: Accessible inside the class and its subclasses.
    - Benefit: Allows subclass modifications without exposing data to the whole program.
    - Drawback: Less strict than private, might allow unintended changes.

Scenario for Protected If designing a base class for different types of bank accounts, a protected interestRate variable would allow subclasses like SavingsAccount to modify it without exposing it publicly.

Impact on Maintenance Encapsulation reduces debugging complexity by preventing unintended data modifications. If everything is public, it becomes harder to track down issues caused by external changes.

Example: If balance in BankAccount were public, another part of the code could modify it directly, bypassing validation, leading to negative balances or incorrect calculations.

Real-World Analogy A vending machine has buttons and internal wiring. Users don't need to see or interact with the wiring; they only press buttons to get what they need. Keeping the wiring hidden prevents tampering and errors.

Part B: Small-Class Design (Minimal Coding)

```
class BankAccount {
private:
  double balance;
  int accountNumber;

public:
  BankAccount(int accNum, double initialBalance) {
    accountNumber = accNum;
    balance = initialBalance > 0 ? initialBalance : 0;
  }

  void deposit(double amount) {
    if (amount > 0) {
      balance += amount;
    }
  }

  bool withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
      balance -= amount;
      return true;
    }
    return false;
  }
};
```

Encapsulation Justification

- **Private balance**: Prevents direct modification to ensure all changes go through controlled methods.
- **Private accountNumber**: Should not be altered after account creation.
- **Public deposit()**: Ensures only positive amounts are added.
- **Public withdraw()**: Prevents overdrawing by checking balance before deduction.

Documentation

// Don’t manipulate the balance directly. Use deposit() or withdraw() to ensure validity.

Part C: Reflection & Short-Answer

Pros and Cons

- Benefit 1: Prevents accidental modifications.
- Benefit 2: Makes debugging easier since changes happen in controlled ways.
- Limitation: Adds slight overhead as access must go through methods.

Encapsulation vs. Abstraction

- **Encapsulation**: Hides internal data, exposing only necessary parts.
- **Abstraction**: Hides complex logic, showing only essential functionalities.

Both hide details, but encapsulation protects data, while abstraction simplifies usage.

Testing Encapsulated Classes

- Use public methods to test behavior.
- Write test cases for valid and invalid inputs.
- Mock dependencies if needed.

Part D: Optional Research

Encapsulation in Various Languages

- **Java**: Uses public, private, protected, and package-private (default).
- **C++**: Uses public, private, protected, but lacks package-private.
- Difference: Java enforces strict encapsulation with access modifiers at a package level, whereas C++ relies more on developer discipline.

Encapsulation in Large-Scale Systems Large companies use encapsulation to enforce security, prevent unauthorized access, and ensure consistency. For example, banking systems use encapsulation to restrict access to account balances and transactions, preventing unauthorized changes.