H02 – Homework: Understanding Encapsulation
Kade Miller
Due: 03-25-2025

Part A: Conceptual Questions

1. Define encapsulation in your own words
Encapsulation is when a class keeps its internal data private and only exposes methods for interacting with it. This way, the class controls how data is used and changed. For example, a class might keep a `balance` variable private and only allow updates through a `deposit()` method that checks the amount.

2. Compare public, private, and protected access

Modifier	Benefit	Drawback
`public`	Makes access simple and flexible	Can expose data to misuse
`private`	Best for protecting internal logic	Can make code harder to extend
`protected`	Allows subclasses controlled access	Can lead to tight coupling

> Scenario: Use `protected` when a base class has shared data that should be hidden from outside but available to child classes, like a `health` variable in a game character base class.

3. Why encapsulation helps reduce debugging complexity
When you only allow access through methods, you know exactly where and how a variable can change. That makes it easier to track bugs or spot incorrect usage. You don’t have to hunt across the whole program for direct data changes.

4. Scenario where public data causes problems
If a class exposes a `balance` variable and any part of the code changes it directly, someone might accidentally make it negative or override it without checks. You'd have no control over those changes and it could break the system.

5. Real-world analogy
Think of a vending machine. The buttons and coin slot are the public interface, what you’re allowed to touch. The electronics and motor inside are private, they handle the real work but are hidden. Keeping that private protects the machine from damage and bad input.

Part B: Small-Class Design (C++)

Class Skeleton:

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

public:
   void deposit(double amount);
   bool withdraw(double amount);
};
```

Encapsulation Justification:

- balance is private to prevent direct changes (e.g., setting it negative).
- accountNumber is private because it shouldn't be edited after the account is created.
- deposit() ensures only positive amounts can be added.
- withdraw() can reject overdrafts and make sure balance stays valid.

Documentation Notes:

In documentation, I'd write something like:

> "The balance and account number should never be modified directly. Use only the provided deposit() and withdraw() methods to ensure account integrity."

This sets a clear boundary for developers using the class.

Part C: Reflection & Short-Answer:

Benefits:

- Protects the internal state from accidental or malicious changes
- Makes the code easier to maintain and debug

Limitation:

- Adds some boilerplate (getters/setters) and can slow early development

---

Testing Strategy:

Even if data is private, we can test it by checking behavior:

- Deposit a value, then call a getBalance() method (read-only)
- Try withdrawing more than the balance — the method should return false
- Test edge cases like depositing zero or withdrawing negative amounts

We test what the class does, not what it holds internally.

Part D: Optional Research (Extra Credit):

Encapsulation in Java vs C++

- Java requires getters/setters for most member access. It uses private by default and encourages encapsulation through interfaces.
- C++ allows both direct public access or encapsulation. You have to manually enforce good practices with private.

In both cases, protected works the same, accessible to subclasses.